Input Generation for SQL Injection and Cross-Site Scripting Exploits

Written By: Adam Kiezun /MIT, Philip Guo /Stanford, Karthick Jayaraman /U. Syracuse, Michael Ernst /U. Washington

Presenter: Corey Buckley

(2008)
Can see that it was most popular 4 years after initial publishing. Since 2016, has started to die down.
The Problem

- Many web applications are widely available with complicated infrastructure supporting them -- allowing malicious users a variety of vectors to exploit.
- From around the time of the paper, ~ 2008, it states “up to 70% of recently reported vulnerabilities affected web applications”
- Websites are inherently more suited to dynamic analysis due to the unpredictability of user input and variety of execution environments (browsers, tech stacks, userdata).
SQL Injection (SQLI)

- Exploits lack of validation on user input
- Can execute/alter SQL statements
- Can lead to data leaks/breaches

1' OR '1'='1
First-order XSS (Client side XSS)

- A way for a user to inject JS into another user’s webpage
- Typically has some component of social engineering

Check out this link! ;) https://www.securebank.com/?username=<script src='mywebsite.com/mybadscript'></script>
Second-order XSS (Server side XSS)

- Attacker is able to inject JS to many users.
- Eliminates the need of social engineering
Groundwork: Sources & Sinks

- A **source** is any user input data. In the case of Ardilla, things like $_GET
- A **sensitive sink** is native PHP function where tainted input could be dangerous, like, mysql_query(), echo

```plaintext
{ Password: '1' OR '1'='1
}
```
Vulnerable Program

Red = tainted

Pink = taint to sensitive sink flow

Example web forum program.

```php
function addMessageForTopic()
{
    if(!isset($_GET['msg']) ||
        !isset($_GET['topicid']) ||
        !isset($_GET['poster'])){
        exit;
    }

    $my_msg = $_GET['msg'];
    $my_topicid = $_GET['topicid'];
    $my_poster = $_GET['poster'];

    //construct SQL statement
    $sqlstmt = "INSERT INTO messages VALUES('".$my_msg.'","'.$my_topicid.'")";

    //store message in database
    $result = mysql_query($sqlstmt);
    echo "Thank you $my_poster for using the message board";
}

function displayAllMessagesForTopic()
{
    if(!isset($_GET['topicid'])){
        exit;
    }

    $my_topicid = $_GET['topicid'];

    $sqlstmt = "SELECT msg FROM messages WHERE topicid='".$my_topicid.'";
    $result = mysql_query($sqlstmt);

    //display all messages
    while($row = mysql_fetch_assoc($result)){
        echo "Message ". $row['msg'];
    }
}
```
SQLI Example

L39: $my_topicid is tainted and flows to L40.
In the case of input like topicid → 1’ OR ‘1’=‘1’
  ○ We’ve injected a tautology into the WHERE clause

```php
$my_topicid = $_GET['topicid'];
$sqlstmt = "SELECT msg FROM messages WHERE topicid='$my_topicid'";
$result = mysql_query($sqlstmt);
```
SQLI Example

```php
//display all messages
while($row = mysql_fetch_assoc($result)){
    echo "Message " . $row['msg'];
}
```

- Now “all messages” really means **ALL** messages.
- A forum could have hidden ‘mod’ topics that are exposed.
First-order XSS Example

```
27  //store message in database
28  $result = mysql_query($sqlstmt);
29  echo "Thank you $my_poster for using the message board";
```

<table>
<thead>
<tr>
<th>mode</th>
<th>→ add</th>
</tr>
</thead>
<tbody>
<tr>
<td>topicid</td>
<td>→ 1</td>
</tr>
<tr>
<td>msg</td>
<td>→ Hello</td>
</tr>
<tr>
<td>poster</td>
<td>→ Villain&lt;script&gt;alert(&quot;XSS&quot;)&lt;/script&gt;</td>
</tr>
</tbody>
</table>

- Consequences?
Second-order XSS Example

```
// construct SQL statement
$sqlstmt = "INSERT INTO messages VALUES('$my_msg','$my_topicid')";

// store message in database
$result = mysql_query($sqlstmt);
```

<table>
<thead>
<tr>
<th>mode</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>topicid</td>
<td>1</td>
</tr>
<tr>
<td>msg</td>
<td>Hello&lt;script&gt;alert(&quot;XSS&quot;)&lt;/script&gt;</td>
</tr>
<tr>
<td>poster</td>
<td>Villain</td>
</tr>
</tbody>
</table>

- Consequences?
What Are Our Options?

- Defensive Coding -- developer’s problem.
- Static Analysis -- can detect patterns, but potentially lots of false positives. Lack of concrete examples.
- Dynamic Analysis -- large runtime overhead.
- White-box testing -- **viable**. Claim it hasn’t discovered unknown vulnerabilities however.
- Black-box testing -- **viable**
Ardilla

Automatic Creation of SQL Injection and Cross-site Scripting (XSS) Attacks

- Ardilla is an automated solution for PHP/MySQL
- White-box tool, i.e. requires source code
- It utilizes a combination of ideas: static analysis, dynamic taint analysis, symbolic execution, etc.
Related Work

- W. Cook and S. Rai. **Safe query objects: statically typed objects as remotely executable queries.** In ICSE, 2005.
- T. Pietraszek and C. V. Berghe. **Defending against injection attacks through context-sensitive string evaluation.** In RAID, 2005.
High Level Overview

- Input generation
  - Taint propagation / executor
    - Attack checker / generator

And a concrete+symbolic database to persist taint.
Architecture Flow, Visualized

Concrete attack vectors!
SQLI & First Order XSS Algorithm

- Where time expired is some arbitrary time constraint.
- Where L4 finds taint sets and a new state of the DB.
- Verifies attack actually produced different DB and new JS in HTML.

(Seems kind of magical, doesn’t it?)

(Assumes the DB is seeded)
Example Of Detection

- Input generator gives input
- We track the propagation of taint
- Each sink has a taint set

(Not sure how they uniquely identify sinks, I’ve used the line number)
Example Of Attack Generation

An example XSS pattern

```plaintext
mode -> add
topicid -> 1
msg -> 1
poster -> <script>alert("XSS")</script>
```
Second Order XSS Algorithm

- Make a symbolic copy of DB so we can persist tainted vars.
- Picks two inputs. *input1* is malicious, *input2* is innocent.
- Compare

```plaintext
parameters: program \( P \), database state \( db \)
result : second-order XSS attack vectors
1  inputs := \( \emptyset \);
2  attacks := \( \emptyset \);
3  \( \text{db}_{sym} := \text{makeSymbolicCopy}(db) \);
4  while not timeExpired() do
5    inputs := inputs \( \cup \) generateNewInput(\( P \));
6    input1 := pickInput(inputs);
7    input2 := pickInput(inputs);
8    \( \langle taints_1, db'_{sym} \rangle := \text{exec\&PropagateTaints}(P, input_1, \text{db}_{sym}) \);
9    \( \langle taints_2, db''_{sym} \rangle := \text{exec\&PropagateTaints}(P, input_2, db'_{sym}) \);
10   attacks := attacks \( \cup \) gen\&CheckAttacks(taints_2, P, \langle input_1, input_2 \rangle);
11  end while
12  return attacks;
```

(Also seems kind of magical, huh?)
Second Order XSS Example

- Naturally, ‘add’ will insert something to our concrete+symbolic DB
- Modifies input 1 to have XSS attack and verifies via input 2.
Input Generation, Black Box

Uses the input generator from Apollo


One of the author’s other works which attempts to find inputs that crash the program.

- Symbolic + Concrete execution
Taint Propagation

- Tracks the data flow of tainted variables.
- A **taint source** is user input function e.g. \$_GET
- **Taint set** describes how a value is affected by input @ runtime
- Strings concatenated with user input have taint set
  - \((\text{$_GET['msg']} + \text{“ by”} + \text{$_GET['poster']})\) ⇒ \{msg, poster\}
- Calls to native PHP functions return union of taint sets for args
Taint Propagation

- Taint sets are stored/fetched in/from the DB when using DB function, E.g. mysql_query
- **Taint filters** are sanitation functions, they return empty taint set
- When a **sensitive sink** has args with non-empty taint sets, then this is a possible vulnerability.
- Augments Zend PHP interpreter to propagate taint
Attack Generator

For those inputs that flow to a sensitive sink,
For each member of the corresponding taint set,
If XSS:
    atks U { ..., arg ← xssAtkLib, ... }
Elif SQLI:
    atks U { ..., arg ← sqlAtkLib, ... }

- Uses a built-in repository of known SQLI & XSS patterns and procedurally generates possible attacks
- 6 SQL, 113 XSS attack patterns

XSS library link was broken :(
Attack Checker

Given a set of atks,

Foreach atk in atks:
  If sql:
    If differentParseTrees(atk, innocentInput):
      Then it’s a SQLI
    Else:
      Atks -= atk # Removing false positive
  Elif xml:
    If htmlScriptChanges(exec(atk), exec(innocentInput)):
      Then it’s an XSS
    Else:
      Atks -= atk
Malicious SQL Parse Tree

SELECT

- COL msg
- FROM TBL messages

WHERE

- Op = COL topicid
- Op = CHAR 1
- Op = CHAR 1

mode → display

topicid → '1' OR '1'='1
Example Second-Order XSS

```html
<html>
    <head>
        ...
    </head>
    <body>
        <script>
            alert("XSS")
        </script>
    </body>
</html>
```
When we load the ‘compromised’ page

This page says

XSS

OK
Concrete & Symbolic Database

- In order to detect Second-Order XSS, we need to persist taint.
- Create a copy of the DB and alter schema
- Insertions are augmented to specify additional columns
- Retrievals are augmented to specify additional columns

Seeded data

<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>

Figure 5: Example state of the concrete+symbolic database table messages used by the PHP program of Figure 1. Each concrete column (left-most two columns) has a symbolic counterpart (right-most two columns) that contains taint sets. The Ø values represent empty taint sets.
Example of SQL Statement Changes

```
INSERT INTO messages VALUES('$_GET['msg']','$_GET['topicid']')
```

```
INSERT INTO messages VALUES('Hello','2', '{msg}', '{topicid}')
```
Example of SQL Statement Changes

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

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

- \( F \) refers to false-positives.
- \( \text{All} \) is sources.
- \( \text{Vuln} \) is verified attacks.

Across these 5 programs, a vulnerability per \( \sim 233 \) lines

XSS1 42\% \( F \) rate in lenient
0\% \( F \) everywhere else
Black Box Fuzzing

- Used 3rd-party tool Burp Intruder on programs except WebChess
  - From paper “1 first-order XSS vulnerability in schoolmate, 3 in faqforge, 0 in EVE, and 0 in geccbbite”
- Those found by Burp Intruder overlapped with Ardilla’s.
- (Not too impressive..)
Limitations

- A sensitive sink may not be reached depending on the input generator. — **Coverage**
- Avg coverage < 50%
- phpBB (35K LOC) had 14% coverage
- One file at a time
- Can’t do sessions
Critique

The Bad

- High XSS1 false positive rate in non-strict mode.
- Implementation not publicly available.
- Didn’t age well (XSS definitions have changed, PHP is not hot anymore)
- said that their attack generation didn't consider subqueries/nested SELECTs
Critique, continued

The Bad

- Does not provide sources for the programs they used to test Ardilla on. Trying to get a program over a decade old working was not fruitful.
- Unable to track program state in other files resulting in low coverage.
- Can only operate on one file at a time
- Felt a little misleading that the paper is called “Input Generation ...” but the actual generation is treated as a black box. (Is actually in the author’s other paper)
Critique, continued

**The Bad**

- Refers to lenient mode to capture “HTML elements like href”. Confusing.
- I had initially thought taint sets were only for sinks for most of the paper.
Critique, continued

The Good

- Ardilla is a precompilation tool -- no runtime cost is incurred.
- Interoperable -- input generator is black box.
- Not much configuration (2 inputs)
- Generates concrete attack vectors that a programmer can verify/test. (can be found at [https://groups.csail.mit.edu/pag/ardilla/](https://groups.csail.mit.edu/pag/ardilla/))
Critique, continued

The Good

- Execution allows for removing false positives in SQLI & XSS2.
- Saving the taint set in the database is a neat idea.
- Enjoyable & easy to read.
- Was influential seeing how many citations it had.
Impact

- Helped with interest in analysis of web applications
  - Interested in client-side code injection in JS
- Shows that there’s many combinations of techniques to use for vulnerability detection.
How The Landscape Has Changed

• In 2020, SQLI has definitive solutions. All used DBMSs I know of support parameterized queries & prepared statements. (Still possible in stored procedures in DSQL b/c of dev concatenating strings, but a dev problem.)

• More intuitive blanket terms for the varying types of XSS: client-side XSS & server-side XSS.

• Common Weakness Enumeration (CWE) had lowered the relevance of SQLI in 2019 to its #6 spot on its top 25. In 2011, was #1. (XSS is currently #4)

• The Open Web Application Security Project (OWASP) top 10 still ranks injection attacks as the most prevalent web vulnerability.
Thanks!

Q/A