Secure web applications

Finding Security Vulnerabilities in Java Applications with Static Analysis

by Livshits & Lam

Presented by Nicholas Kuilema
Overview

- Most web apps use a “safe” language
  - Java, php, etc
- Examine Higher level attacks
  - SQL injection
  - Cross-site scripting
Vulnerabilities

- Inject malicious data
  - Form fields, URLs, HTTP headers, Cookies
- Manipulate applications
  - SQL injection
  - Cross-site scripting
  - HTTP response splitting
  - Path Traversal
  - Command injection
Code Auditing

- Effective
- Time consuming
- Not sound
- Iterate until deadline
Static Analysis to the rescue

- Examines Java Byte code
- Vulnerability patterns described with PQL
- Present results in a GUI – Eclipse
- Analysis is sound – Will not miss bugs*
HttpServletRequest request = ...;
String userName = request.getParameter("name");
Connection con = ...
String query = "SELECT * FROM Users " +
    " WHERE name = '" + userName + "'";
con.execute(query);
SQL in “popular” culture

xkcd.com
Malicious data

- Parameters, URLs, Hidden fields, HTTP headers, Cookies, Non-web inputs
- Never trust the user
- Don’t rely on client side validation
Exploiting Unchecked Input

- Cross Site Scripting
  - Run commands in clients browser
- HTTP Response Splitting
  - Forging/poisoning caches
- Path Traversal
  - Arbitrary file access
- Command Injection
  - Shell access
Static Analysis

- Formalism using Tainted Object Propagation
- Implementation using Datalog
Tainted Object Propagation

- access path: Sequence of field accesses, array index operations, or method calls separated by dots. $v.f.g$ or $\epsilon$

- Source descriptor $\langle m, n, p \rangle$
  
  $m$ - method
  
  $n$ - parameter number (-1 is return value)
  
  $p$ - access path
Tainted Object Propagation

Sink Descriptors \(\langle m, n, p \rangle\)
- \(m\) - method
- \(n\) - parameter number (-1 is return value)
- \(p\) - access path

Derivation Descriptors \(\langle m, n_s, p_s, n_d, p_d \rangle\)
- \(m\) - method
- \(n_s\) - source number
- \(p_s\) - source path
- \(n_d\) - destination number
- \(p_d\) - destination path
Derivation Descriptors

- Java Strings are immutable
- Taint must be passed to each string created from manipulation
- Unicode, UTF-8, URL encoding
- No descriptors for sanitization routines
Security Violation

- Source object
- Sink object
- Derived(01,02)

Objects are tainted if obtained from a Derived chain of length $\geq 0$ from a source object

Security Violation if a Sink object is tainted
Tainted Object Propagation Specification

- Manually examined J2EE APIs
- Instrumented calls to application code
- Static analysis to locate methods that take tainted objects as arguments
- No proof the specification is complete*
Static Analysis

- Requires points-to analysis to limit taint from propagating everywhere
- Uses a finite set of static object "Names"
- \texttt{pointsto(v,h)}
  - \(v\) – program variable
  - \(h\) – allocation site
**Definition 3.6** A *static security violation* is a sequence of heap allocation sites \( h_1 \ldots h_k \) such that

1. There exists a variable \( v_1 \) such that \( \text{pointsto}(v_1, h_1) \), where \( v_1 \) corresponds to access path \( p \) applied to argument \( n \) of a call to method \( m \) for a source descriptor \( \langle m, n, p \rangle \).

2. There exists a variable \( v_k \) such that \( \text{pointsto}(v_k, h_k) \), where \( v_k \) corresponds to applying access path \( p \) to argument \( n \) in a call to method \( m \) for a sink descriptor \( \langle m, n, p \rangle \).

3. There exist variables \( v_1, \ldots, v_k \) such that

\[
\forall 1 \leq i < k : \text{pointsto}(v_i, h_i) \land \text{pointsto}(v_{i+1}, h_{i+1}),
\]

where variable \( v_i \) corresponds to applying \( p_s \) to argument \( n_s \) and \( v_{i+1} \) corresponds applying \( p_d \) to argument \( n_d \) in a call to method \( m \) for a derivation descriptor \( \langle m, n_s, p_s, n_d, p_d \rangle \).
Static Analysis

- Based on Whaley and Lam
- Context-sensitive
- Uses binary decision diagrams (BDDs)
- Results accessed through Datalog
- Turn violation definition into a query
Scaling

- Preciseness needed for a useful tool
- Context sensitive can hurt scaling
- BDDs allow $10^{14}$ contexts
- Allows for almost 1000 classes
Program Query Language (PQL)

- Syntactic sugar for Datalog queries
- Simple translation to Datalog
- Java-like syntax
- ; sequence
- _ wildcard
- | alternative

```java
query main()
returns
object Object sourceObj, sinkObj;
matches {
    sourceObj := source();
    sinkObj := derived*(sourceObj);
    sinkObj := sink();
}
```
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.execute(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(_)
  | ...
}

query derived*(object Object x)
returns
  object Object y;
uses
  object Object temp;
matches {
  y := x |
  temp := derived(x); y := derived*(temp);
}
Precision Improvements

- Containers
  - Name collisions in alias analysis
- String Routines
  - String.toLowerCase()
Eclipse GUI
Experiments

9 apps
- 8 real
- 1 synthetic

140 - 52K LoC

264 - 989 Classes
<table>
<thead>
<tr>
<th>Context sensitivity</th>
<th>Pre-processing</th>
<th>Points-to analysis</th>
<th>Taint analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved naming</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>
# Tainted Objects

<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><strong>jboard</strong></td>
<td>1 6</td>
<td>268 23 2 2</td>
<td>0 0 2 2</td>
<td>2 2</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td><strong>blueblog</strong></td>
<td>6 12</td>
<td>17 17 17 17</td>
<td>1 1 1 1</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td><strong>webgoat</strong></td>
<td>13 59</td>
<td>1,166 201 903 157</td>
<td>51 7 51 6</td>
<td>45 1 45 0</td>
<td>20 1 81 0</td>
<td>6</td>
</tr>
<tr>
<td><strong>blojsom</strong></td>
<td>27 18</td>
<td>368 203 197 112</td>
<td>48 4 26 2</td>
<td>46 2 24 0</td>
<td>17 11 15 0</td>
<td>2</td>
</tr>
<tr>
<td><strong>personalblog</strong></td>
<td>25 31</td>
<td>2,066 1,023 1,685 426</td>
<td>460 275 370 2</td>
<td>48 1 47 0</td>
<td>458 2 73 0</td>
<td>2</td>
</tr>
<tr>
<td><strong>snipsnap</strong></td>
<td>155 100</td>
<td>1,168 791 897 456</td>
<td>732 93 313 27</td>
<td>717 78 498 12</td>
<td>38 1 38 0</td>
<td>15</td>
</tr>
<tr>
<td><strong>road2hibernate</strong></td>
<td>1 33</td>
<td>2,150 843 1,641 385</td>
<td>18 12 16 1</td>
<td>17 1 17 0</td>
<td>17 11 15 0</td>
<td>1</td>
</tr>
<tr>
<td><strong>pebble</strong></td>
<td>132 70</td>
<td>1,403 621 957 255</td>
<td>427 211 193 1</td>
<td>426 210 192 0</td>
<td>17 11 15 0</td>
<td>1</td>
</tr>
<tr>
<td><strong>roller</strong></td>
<td>32 64</td>
<td>2,367 504 1,923 151</td>
<td>378 12 261 1</td>
<td>377 11 260 0</td>
<td>17 11 15 0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total**

392 393 10,973 4,226 8,222 1,961 2,115 615 1,431 41 2,086 586 1,402 12 29

![Number of tainted objects](graph.png)

**Legend:**
- **Context-insensitive, default naming**
- **Context-insensitive, improved naming**
- **Context-sensitive, default naming**
- **Context-sensitive, improved naming**
## Vulnerability Types

<table>
<thead>
<tr>
<th>Sinks</th>
<th>SQL injections</th>
<th>HTTP splitting</th>
<th>Cross-site scripting</th>
<th>Path traversal</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header manip.</td>
<td></td>
<td></td>
<td>snipsnap = 6</td>
<td>blueblog: 1, webgoat: 1, pebble: 1, roller: 1 = 4</td>
<td>0</td>
</tr>
<tr>
<td>Parameter manip.</td>
<td>webgoat: 4, personalblog: 2 = 6</td>
<td>snipsnap = 5</td>
<td>0</td>
<td>blojsom = 2</td>
<td>0</td>
</tr>
<tr>
<td>Cookie poisoning</td>
<td>webgoat = 1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Non-Web inputs</td>
<td>snipsnap: 1, road2hibernate: 1 = 2</td>
<td>0</td>
<td></td>
<td>snipsnap = 3</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>9</td>
<td>11</td>
<td>4</td>
<td>5</td>
<td>29</td>
</tr>
</tbody>
</table>

**Figure 11:** Classification of vulnerabilities we found. Each cell corresponds to a combination of a source type (in rows) and sink type (in columns).
Vulnerabilities

- 29 found - 26 unique
- snipsnap accounts for over half
- snipsnap accounts for all HTTP splitting
- blojsom has bad sanitization, but fixing it would still produce a false positive
False Positives

- All in snipsnap
- Distinguishing return values for `StringWriter.toString()` eliminates them all
Questions?