Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks

Alex Orso
with William Halfond and Pete Manolios
Georgia Institute of Technology
{orso|whalfond|manolios}@cc.gatech.edu

**SQL Injection**

```java
String queryString = "SELECT info FROM userTable WHERE ";
if (!login.equals("")) && (!password.equals("")) {
    queryString += "login='" + login + "' AND pass='" + password + "';
} else {
    queryString+="login='guest'";
}
ResultSet tempSet = stmt.executeQuery(queryString);
```
Attack Scenario

```
String queryString = "SELECT info FROM userTable WHERE ";
if (!(login.equals("")) && (! password.equals(""))) {
    queryString += "login='' + login + "' AND pass='" + password + ";";
} else {
    queryString+="login='guest'';
}
ResultSet tempSet = stmt.executeQuery(queryString);
```

Normal Usage
User submits login “doe” and password “xyz"

```
SELECT info FROM users WHERE login='doe' AND pass='xyz'
```
Attack Scenario

String queryString = "SELECT info FROM userTable WHERE ";
if (!(login.equals("")) && (! password.equals(""))) {
    queryString += "login='" + login + "' AND pass='" + password + ";
} else {
    queryString += "login='guest'";
}
ResultSet tempSet = stmt.executeQuery(queryString);

Malicious Usage
Attacker submits “admin' -- " and password of ""

SELECT info FROM users WHERE login='admin' -- ' AND pass="
Overall Goal of The Project

- Protecting existing (insecure) Web applications by automatically detecting and preventing SQLIAs

- Highly automated — Little/no human effort
- Conservative — No false negatives
- Precise — Few/no false positives
WASP
(Web Application SQL-injection Preventer)

Basic idea => Allow only developer-trusted strings to form sensitive parts of a query

Solution:
1. Positive tainting
2. Syntax-Aware Evaluation
public Login(request, response) {
String login = request.getParameter("login");
String pin = request.getParameter("pin");
Statement stmt = connection.createStatement();
String queryString = "SELECT info FROM userTable WHERE ";
if (! login.equals("")) && (! password.equals("")) {
    queryString += "login='" + login + "' AND pass='" + password + "'";
} else {
    queryString+="login='guest'";
}
ResultSet tempSet = stmt.executeQuery(queryString);
}
public Login(request, response) {
    String login = request.getParameter("login");
    String pin = request.getParameter("pin");
    Statement stmt = connection.createStatement();
    String queryString = "SELECT info FROM userTable WHERE ";
    if (! login.equals("") && ! password.equals("")) {
        queryString += "login='" + login + "' AND pass='" + password + "';";
    } else {
        queryString += "login='guest'";
    }
    ResultSet tempSet = stmt.executeQuery(queryString);
}
Positive Tainting

Identify and mark **trusted** data instead of untrusted data

=> **Increased automation:** Trusted data readily identifiable in Web applications

=> **Increased safety:** Incompleteness leads to easy-to-eliminate false positives
(normal in-house testing causes set of trusted data to converge to complete set)

In general, it implements the security principle of “fail-safe defaults”
Syntax-aware Evaluation

• Cannot simply forbid the use of untrusted data in queries
• Dependence on filtering rules requires unsafe assumptions

=> Syntax-aware evaluation
  • Performed right before the query is sent to the database
  • Consider the context in which trusted and untrusted data is used: permit untrusted data to be only in string and numeric literals
Example

1. String queryString = "SELECT info FROM userTable WHERE ";
2. if (! login.equals("")) && (! password.equals("")) {
3.    queryString += "login='" + login + "' AND pass='" + password + "';"
4. } else {
5.    queryString+="login='guest'";
}
6. ResultSet tempSet = stmt.executeQuery(queryString);

login -> “doe”, password -> “xyz”

MetaString: queryString

[WH][E][R][E][l][o][g][i][n][=]\[d][o][e]\[A][N][D]\[p][a][s][s][=]\[x][y][z]\[=\]

SELECT info FROM userTable WHERE login = 'doe' AND pass = 'xyz'

[ ] == trusted
Example

1. String queryString = "SELECT info FROM userTable WHERE ";
2. if (!(login.equals("")) && (! password.equals(""))) {
3.    queryString += "login='" + login + "' AND pass='" + password + "'";
    }
4. else {
    queryString += "login='guest'";
    }
5. ResultSet tempSet = stmt.executeQuery(queryString);

login -> "admin' -- ", password -> ""

MetaString: queryString

```
... [E][R][E][l][o][g][i][n][e]=[']a][d][m][i][n][']][][-][-][']A][N][D][p][a][s][s][s]=[']"'
```

\[ \] == trusted
Tool Implementation (Java + Tomcat)
Tool Implementation (Java + Tomcat)

Minimal deployment requirements

- No need for a customized runtime system (based on on-line instrumentation)
- Highly automated
- Transparent for the system administrator
Evaluation

1. Effectiveness and accuracy
   1. **False negatives**: How many attacks go undetected?
   2. **False positives**: How many legitimate accesses are blocked as attacks?

2. **Overhead**: What is the runtime cost of using WASP?
Experiment Setup

<table>
<thead>
<tr>
<th>Subject</th>
<th>LOC</th>
<th>Database Interaction Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkers</td>
<td>5,421</td>
<td>5</td>
</tr>
<tr>
<td>Office Talk</td>
<td>4,543</td>
<td>40</td>
</tr>
<tr>
<td>Employee Directory</td>
<td>5,658</td>
<td>23</td>
</tr>
<tr>
<td>Bookstore</td>
<td>16,959</td>
<td>71</td>
</tr>
<tr>
<td>Events</td>
<td>7,242</td>
<td>31</td>
</tr>
<tr>
<td>Classifieds</td>
<td>10,949</td>
<td>34</td>
</tr>
<tr>
<td>Portal</td>
<td>16,453</td>
<td>67</td>
</tr>
</tbody>
</table>

- Applications are a mix of commercial (5) and student projects (2)
- Attacks and legitimate inputs developed *independently*
- Attack inputs represent broad range of exploits
Evaluation Results: Effectiveness/Accuracy

<table>
<thead>
<tr>
<th>Subject</th>
<th># Legit. Accesses</th>
<th>False Positives</th>
<th>Total # Attacks</th>
<th>Successful Attacks</th>
<th>Original Web Apps</th>
<th>WASP Protected Web Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkers</td>
<td>1,359</td>
<td>0</td>
<td>4,431</td>
<td>922</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Office Talk</td>
<td>424</td>
<td>0</td>
<td>5,888</td>
<td>499</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Empl. Dir</td>
<td>658</td>
<td>0</td>
<td>6,398</td>
<td>2,066</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bookstore</td>
<td>607</td>
<td>0</td>
<td>6,154</td>
<td>1,999</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Events</td>
<td>900</td>
<td>0</td>
<td>6,207</td>
<td>2,141</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Classifieds</td>
<td>574</td>
<td>0</td>
<td>5,968</td>
<td>1,973</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Portal</td>
<td>1,080</td>
<td>0</td>
<td>6,403</td>
<td>3,016</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

No false positives or false negatives in our evaluation.
### Evaluation Results: Effectiveness/Accuracy

<table>
<thead>
<tr>
<th>Subject</th>
<th># Legit. Accesses</th>
<th>False Positives</th>
<th>Total # Attacks</th>
<th>Successful Attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkers</td>
<td>1,359</td>
<td>0</td>
<td>4,431</td>
<td>922</td>
</tr>
<tr>
<td>Office Talk</td>
<td>424</td>
<td>0</td>
<td>5,888</td>
<td>499</td>
</tr>
<tr>
<td>Empl. Dir</td>
<td>658</td>
<td>0</td>
<td>6,398</td>
<td>2,066</td>
</tr>
<tr>
<td>Bookstore</td>
<td>607</td>
<td>0</td>
<td>6,154</td>
<td>1,999</td>
</tr>
<tr>
<td>Events</td>
<td>900</td>
<td>0</td>
<td>6,207</td>
<td>2,141</td>
</tr>
<tr>
<td>Classifieds</td>
<td>574</td>
<td>0</td>
<td>5,968</td>
<td>1,973</td>
</tr>
<tr>
<td>Portal</td>
<td>1,080</td>
<td>0</td>
<td>6,403</td>
<td>3,016</td>
</tr>
</tbody>
</table>

No false positives or false negatives in our evaluation.
Evaluation Results: Overhead

<table>
<thead>
<tr>
<th>Subject</th>
<th># Inputs</th>
<th>Avg. Access Time (ms)</th>
<th>Avg. Access Overhead (ms)</th>
<th>% Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkers</td>
<td>1,359</td>
<td>122</td>
<td>5</td>
<td>5%</td>
</tr>
<tr>
<td>Office Talk</td>
<td>424</td>
<td>56</td>
<td>1</td>
<td>2%</td>
</tr>
<tr>
<td>Empl. Dir</td>
<td>658</td>
<td>63</td>
<td>3</td>
<td>5%</td>
</tr>
<tr>
<td>Bookstore</td>
<td>607</td>
<td>70</td>
<td>4</td>
<td>6%</td>
</tr>
<tr>
<td>Events</td>
<td>900</td>
<td>70</td>
<td>1</td>
<td>1%</td>
</tr>
<tr>
<td>Classifieds</td>
<td>574</td>
<td>70</td>
<td>3</td>
<td>5%</td>
</tr>
<tr>
<td>Portal</td>
<td>1,080</td>
<td>83</td>
<td>16</td>
<td>19%</td>
</tr>
</tbody>
</table>

Overhead is dominated by network and database access time.
Conclusions and Future Work

WASP: Technique for securing applications against SQL injection attacks

Advantages
• Highly/fully automated
• Effective and accurate
• Minimal deployment requirements

Evaluation involving over 47,000 accesses showed no false positives or false negatives

Future work
• Static analysis to optimize dynamic instrumentation
• Apply general principle to other forms of attacks
• Commercialization
Thanks!

Questions?