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

William G.J. Halfond
Alessandro Orso
Panagiotis Manolios

Georgia Institute of Technology

Deployment context of a typical Web application.
Deployment context of a typical Web application.
SQL Injection Attacks

Easy to create a database query – hard to do it securely.

• Open Web Application Security Project (OWASP) lists SQLIA in its top ten most critical web application security vulnerabilities
• David Aucsmith (CTO of Security and Business Unit, Microsoft) defined SQLIA as one of the most serious threats to web apps
• Successful attacks on Guess Inc., Travelocity, FTD.com, Tower Records, RIAA, …
• Companies have built their business on detecting SQLIAs
Example of an SQLIA

```java
public Login(request, response) {
    String login = request.getParameter("login");
    String passwd = request.getParameter("passwd");
    String query = "SELECT info FROM userTable WHERE ";
    if (!login.equals("")) && (!password.equals(""))
        query += "login='"+login+'" AND pass='"+passwd +""
    else
        query+="login='guest'";
    ResultSet result = stmt.executeQuery(query);
    if (result != null)
        displayAccount(result);
    else
        sendAuthFailed();
}
```
Example of an SQLIA

```java
public Login(request, response) {
    String login = request.getParameter("login");
    String passwd = request.getParameter("passwd");
    String query = "SELECT info FROM userTable WHERE ";
    if (!login.equals("")) && (!password.equals(""))
        query += "login='"+login+"' AND pass='"+passwd +"'
    else
        query+="login='guest'";
    ResultSet result = stmt.executeQuery(query);
    if (result != null)
        displayAccount(result);
    else
        sendAuthFailed();
}
```

Normal Usage
- User submits login “**doe**” and passwd “**xyz**”
- `SELECT info FROM users WHERE login='doe' AND pass='xyz'`
Example of an SQLIA

```java
public Login(request, response) {
    String login = request.getParameter("login");
    String passwd = request.getParameter("passwd");
    String query = "SELECT info FROM userTable WHERE ";
    if (! login.equals("")) && (! password.equals(""))
        query += "login='"+login+"' AND pass='"+
    else
        query+="login='guest'";
    ResultSet result = stmt.executeQuery(query);
    if (result != null)
        displayAccount(result);
    else
        sendAuthFailed();
}
```

**Malicious Usage**

- Attacker submits “admin’ -- ” and passwd of “0”
- `SELECT info FROM users WHERE login='admin' -- ' AND pass='0'`
Presentation Outline

- Our Technique
  - Positive tainting
  - Syntax-aware evaluation
- Implementation -- WASP
- Evaluation
- Related work
- Conclusions and future work
Our Technique

Basic approach => Only allow developer-trusted strings to form sensitive parts of a query

Solution:

1. **Positive tainting**: Identify and mark developer-trusted strings. Propagate taint markings at runtime

2. **Syntax-Aware Evaluation**: Check that all keywords and operators in a query were formed using marked strings
Example: Positive vs. Negative Tainting

```java
public Login(request, response) {
    String login = request.getParameter("login");
    String passwd = request.getParameter("passwd");
    String query = "SELECT info FROM userTable WHERE ";
    if (! login.equals("")) && (! password.equals(""))
        query += "login='"+login+"' AND pass='"+passwd + "'
    else
        query+="login='guest'";
    ResultSet result = stmt.executeQuery(query);
    if (result != null)
        displayAccount(result);
    else
        sendAuthFailed();
}
```

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

Negative tainting.  Positive tainting.
Benefits of Positive Tainting

⇒ Increased safety: Incompleteness leads to easy-to-eliminate false positives
⇒ Normal in-house testing causes set of trusted data to converge to complete set
⇒ Implements security principle of “fail-safe defaults” [Saltzer and Schroeder]
⇒ Increased automation: Trusted data readily identifiable in Web applications
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
Complete Example

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

login -> “doe”, password -> “xyz”
Complete Example

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

login -> “doe”, password -> “xyz”

queryString
[S][E][L][E][C][T] info FROM [u][s][e][r][T][a][B][L][E] [W][H][E][R][E] [;]

[W][H][E][R][E]
Complete Example

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

login -> “doe”, password -> “xyz”

```
queryString
[S][E][L][E][C][T] [i][n][f][o] [F][R][O][M] [u][s][e][r][T][a][b][l][e] [W][H][E][R][E]
```

```
<table>
<thead>
<tr>
<th>tmp0</th>
<th>tmp1</th>
<th>tmp2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="image1" /></td>
<td><img src="image2.png" alt="image2" /></td>
<td><img src="image3.png" alt="image3" /></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>tmp3</th>
<th>tmp4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4.png" alt="image4" /></td>
<td><img src="image5.png" alt="image5" /></td>
</tr>
</tbody>
</table>
```

William Halfond – FSE 2006 – November 8th, 2006 – Slide 15
### Complete Example

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

login -> “doe”, password -> “xyz”

```sql
queryString
...[W][H][E][R][E][t][h][e][r][o][g][i][n]="doe"[A][N][D][p][a][s][s]="xyz"
```
Complete Example

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

login -> “doe”, password -> “xyz”

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

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

login -> “doe”, password -> “xyz”
Complete Example

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

login -> “admin’ -- “, password -> “”

queryString
... [R][E][][][o][g][i][n][=]['] [a][d][m][i][n]['=]['] [A][N][D][p][a][s][s][=][']
Complete Example

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

login -> “admin’ -- " , password -> “"

SELECT info FROM userTable WHERE login='admin' -- ' AND pass=’
Complete Example

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

login -> “admin’ -- “, password -> “"
WASP Architecture
Tracking the Taint Markings

⇒ MetaStrings: library that mimics all string-related classes

Benefits of the approach:
1. Complete mediation of all string operations
2. Polymorphism reduces instrumentation.
3. Track at the right level of granularity: character-level tainting
Implementation: Positive Tainting

• Identify developer-trusted strings.
  1. Hard-coded strings
  2. Implicitly-created strings
  3. Strings from external sources
• Use instrumentation to:
  1. Replace with MetaStrings
  2. Assign trust markings
Minimal Deployment Requirements

- No need for a customized runtime system
- Based on instrumentation
  - Off-line
  - On the fly
- Highly automated
- Transparent for the system administrator
Evaluation

1. False negatives: How many attacks go undetected?
2. False positives: How many legitimate accesses are blocked as attacks?
3. 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: 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></td>
</tr>
<tr>
<td>Office Talk</td>
<td>424</td>
<td>0</td>
<td>5,888</td>
<td>499</td>
<td>0</td>
<td></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></td>
</tr>
<tr>
<td>Bookstore</td>
<td>607</td>
<td>0</td>
<td>6,154</td>
<td>1,999</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td>900</td>
<td>0</td>
<td>6,207</td>
<td>2,141</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Classifieds</td>
<td>574</td>
<td>0</td>
<td>5,968</td>
<td>1,973</td>
<td>0</td>
<td></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></td>
</tr>
</tbody>
</table>
# Evaluation Results: 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></td>
<td></td>
<td></td>
<td></td>
<td>Original Web Apps</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WASP Protected Web Apps</td>
</tr>
<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>
## Evaluation Results: 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></td>
<td>922</td>
<td>0</td>
</tr>
<tr>
<td>Office Talk</td>
<td>424</td>
<td>0</td>
<td>5,888</td>
<td></td>
<td>499</td>
<td>0</td>
</tr>
<tr>
<td>Empl. Dir</td>
<td>658</td>
<td>0</td>
<td>6,398</td>
<td></td>
<td>2,066</td>
<td>0</td>
</tr>
<tr>
<td>Bookstore</td>
<td>607</td>
<td>0</td>
<td>6,154</td>
<td></td>
<td>1,999</td>
<td>0</td>
</tr>
<tr>
<td>Events</td>
<td>900</td>
<td>0</td>
<td>6,207</td>
<td></td>
<td>2,141</td>
<td>0</td>
</tr>
<tr>
<td>Classifieds</td>
<td>574</td>
<td>0</td>
<td>5,968</td>
<td></td>
<td>1,973</td>
<td>0</td>
</tr>
<tr>
<td>Portal</td>
<td>1,080</td>
<td>0</td>
<td>6,403</td>
<td></td>
<td>3,016</td>
<td>0</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.
Related Work

Similar Dynamic Tainting Approaches

- Nguyen-Tuong et. al.
- Pietraszek and Berghe

Other Dynamic Tainting Approaches

- Haldar, Chandra, and Franz
- Martin, Livshits, and Lam

Other approaches discussed in the paper.
Conclusions and Future Work

- **WASP**: Highly automated technique for securing applications against SQL Injection Attacks
  - Positive tainting
  - Accurate and efficient taint propagation
  - Syntax-aware evaluation
  - Minimal deployment requirements
- Evaluation involving over 47,000 web accesses showed no false positives or false negatives
- Future work
  - Use static analysis to optimize dynamic instrumentation
  - Apply general principle to other forms of attacks
Questions?