



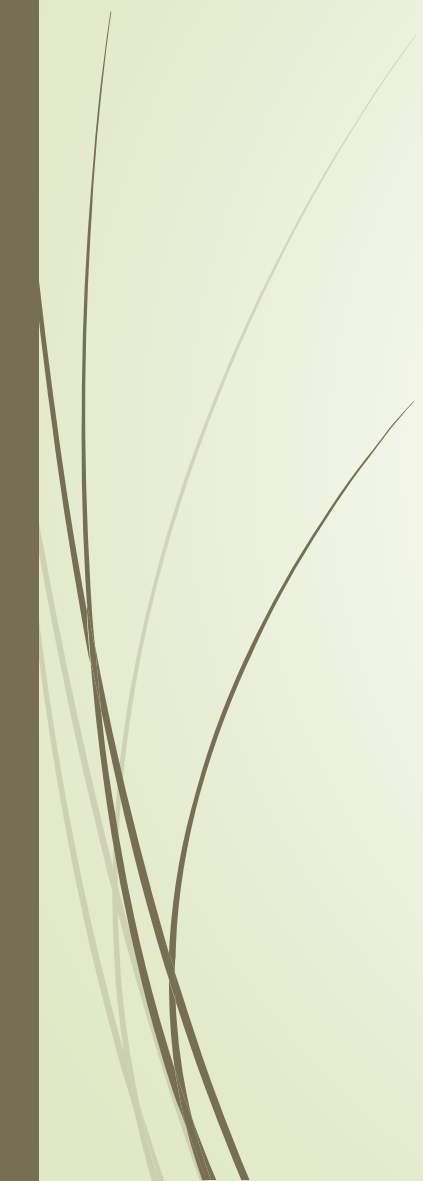
WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation

William G.J. Halfond, Alessandro Orso, Panagiotis Manolios

Published in TSE, 2008



Outline

- Motivation
 - Approach
 - Implementation
 - Evaluation
 - Conclusion
- 



Motivation



SQLIA



SQLIA



What



When



How



Main
variants

In general, SQLIAs are a class of code injection attacks that take advantage of the lack of validation of user input.



SQLIA



These attacks occur when developers combine hard-coded strings with user-provided input to create dynamic queries.



SQLIA



What

When

How

Main
variants

If user input is not properly validated, attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings.

SQLIA

What

When

How

Main variants

Example:



```
1. String login = getParameter("login");
2. String pin = getParameter("pin");
3. Statement stmt = connection.createStatement();
4. String query = "SELECT acct FROM users WHERE login='";
5. query += login + "' AND pin=" + pin;
6. ResultSet result = stmt.executeQuery(query);
7. if (result != null)
8.     displayAccount(result); // Show account
9. else
10.    sendAuthFailed(); // Authentication failed
```

Fig. 2. Excerpt of a Java servlet implementation.

```
SELECT acct FROM users WHERE login='doe' AND pin=123
```

```
SELECT acct FROM users WHERE login='admin' -- ' AND pin=0
```



SQLIA



What

When

How

Main
variants

- Tautologies
- Union Queries
- Piggybacked Queries



Approach

- Positive tainting
 - Syntax-aware evaluation
- 



Taint checking

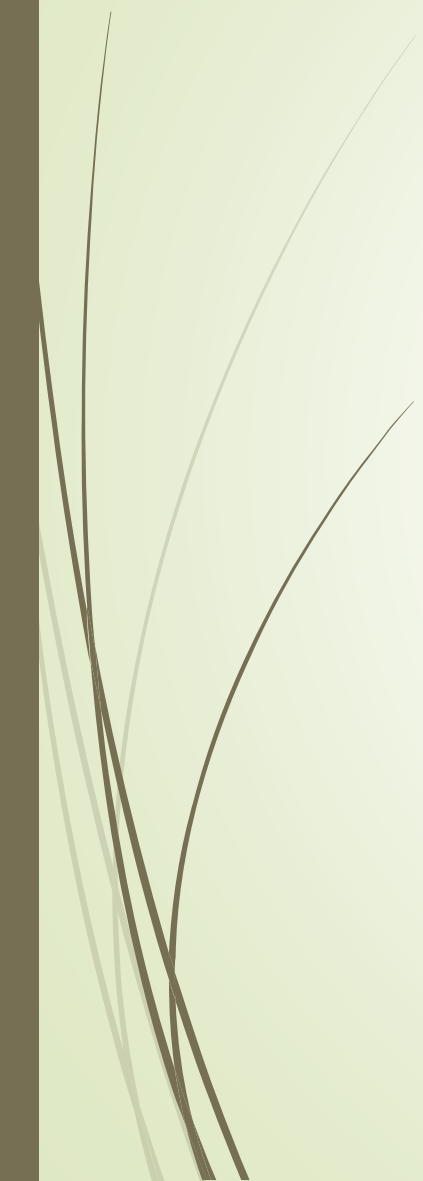
- A feature in some computer programming languages, designed to increase security by preventing malicious users from executing commands on a host computer.
- The concept behind taint checking is that any variable that can be modified by an outside user poses a potential security risk.



Positive tainting



Differences

- It is based on the identification, marking, and tracking of trusted, rather than untrusted, data.
- 

Positive tainting

Differences

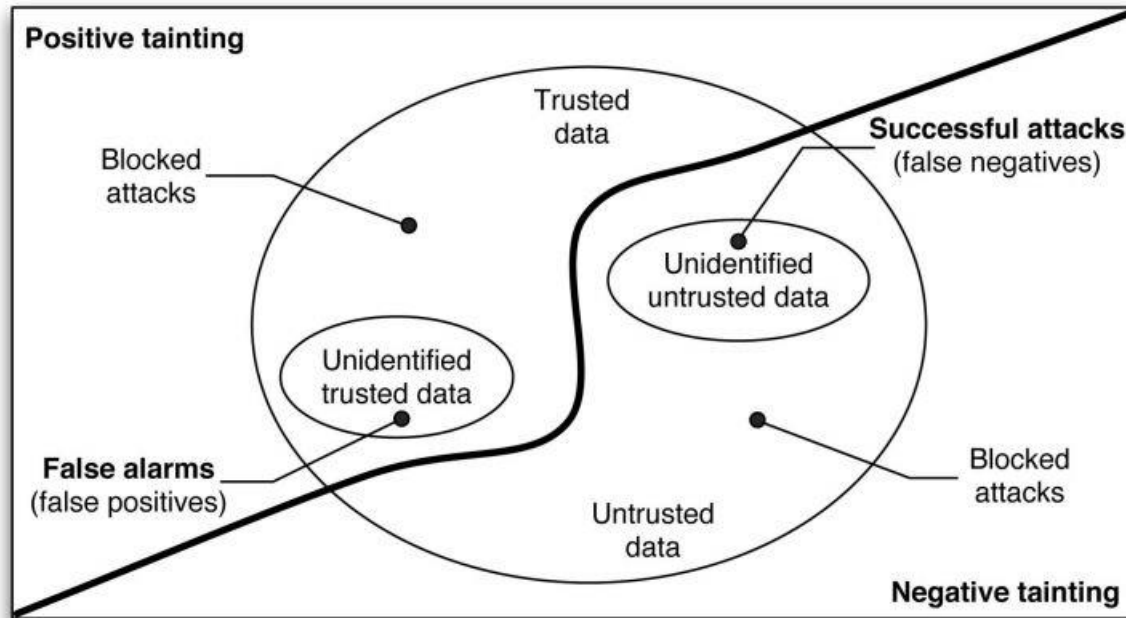


Fig. 3. Identification of trusted and untrusted data.



Syntax-aware evaluation

- It considers the context in which trusted and untrusted data is used to make sure that all parts of a query other than string or numeric literals consist only of trusted characters.
- As long as untrusted data is confined to literals, no SQLIA can be performed.

Syntax-aware evaluation

- 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
... [W][H][E][R][E][ ][!][o][g][i][n][=][ ][d][o][e][ ][A][N][D][ ][p][a][s][s][=][ ][x][y][z][ ]
```

Syntax-aware evaluation

- 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][I][O][G][I][N]=[][a][d][m][i][n]'[-][-][][][A][N][D][][p][a][s][s]=[][]'

Implementation

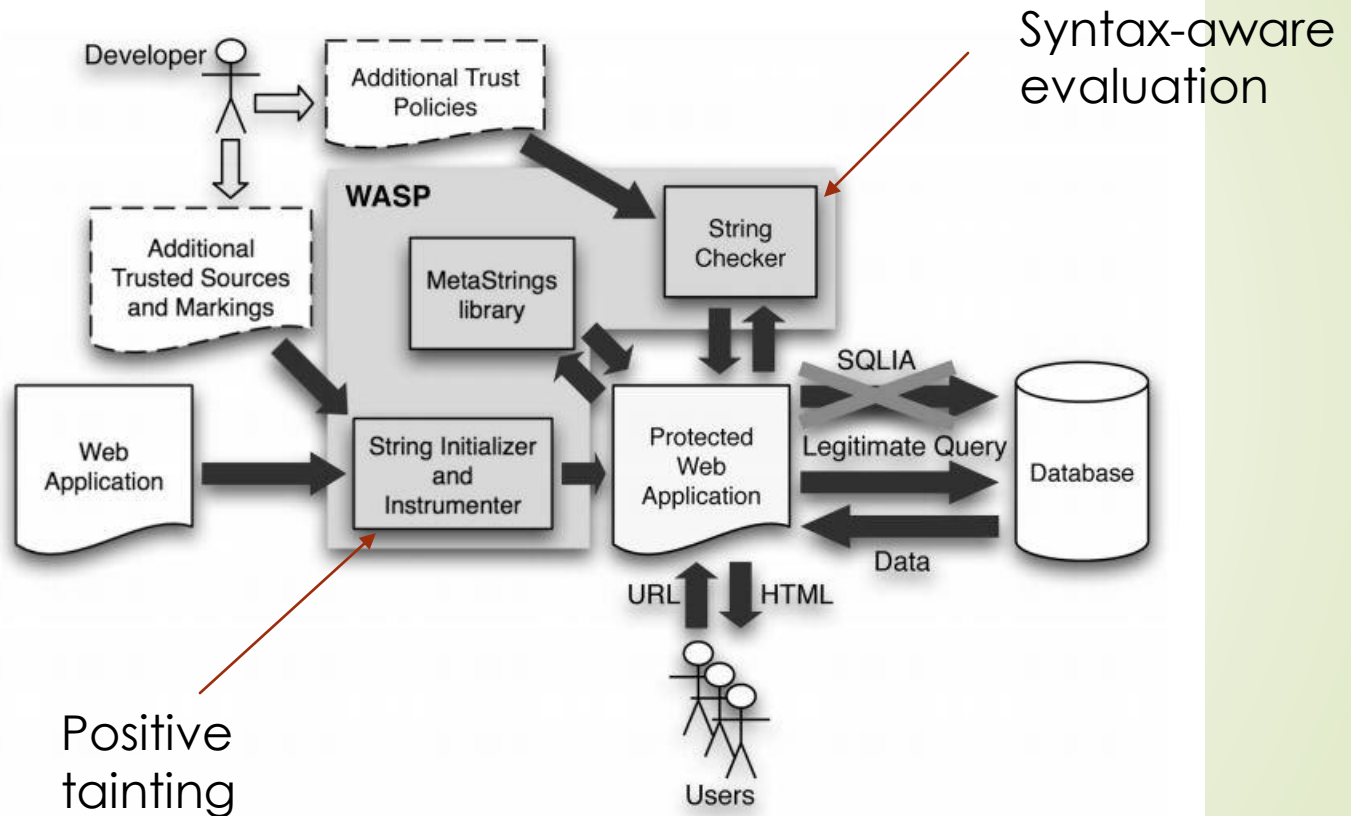


Fig. 4. High-level overview of the approach and tool.



Evaluation

- RQ1. What percentage of attacks can WASP detect and prevent that would otherwise go undetected and reach the database?
- RQ2. What percentage of legitimate accesses are incorrectly identified by WASP as attacks?
- RQ3. What is the runtime overhead imposed by WASP on the Web applications that it protects?

Evaluation

- RQ1. What percentage of attacks can WASP detect and prevent that would otherwise go undetected and reach the database?

TABLE 2
Results of Testing for False Negatives (RQ1)

<i>Subject</i>	<i>Total # Attacks</i>	<i>Successful Attacks</i>	
		<i>Original Web Apps</i>	<i>WASP Protected Web Apps</i>
Checkers	4,431	922	0
Office Talk	5,888	499	0
Empl. Dir.	6,398	2,066	0
Bookstore	6,154	1,999	0
Events	6,207	2,141	0
Classifieds	5,968	1,973	0
Portal	6,403	3,016	0
Daffodil	19	19	0
Filelister	96	80	0
WebGoat	96	88	0

Evaluation

- RQ2. What percentage of legitimate accesses are incorrectly identified by WASP as attacks?

TABLE 3
Results of Testing for False Positives (RQ2)

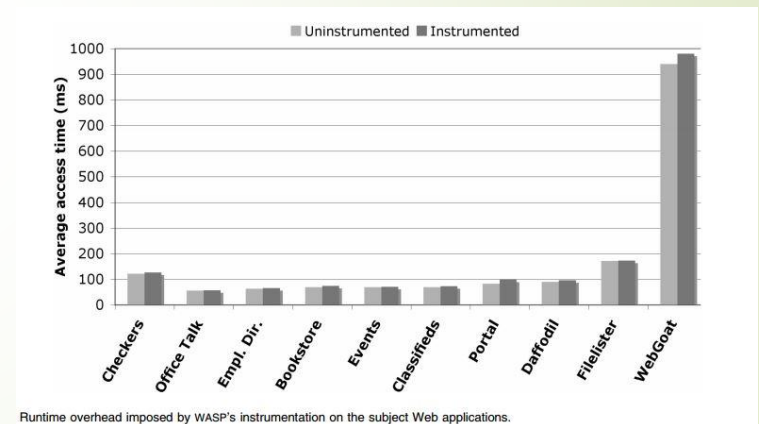
<i>Subject</i>	<i># Legitimate Accesses</i>	<i>False Positives</i>
Checkers	1,359	0
Office Talk	424	0
Empl. Dir.	1,244	0
Bookstore	3,239	0
Events	1,324	0
Classifieds	2042	0
Portal	3,435	0
Daffodil	19	0
Filelister	40	0
WebGoat	40	0

Evaluation

- RQ3. What is the runtime overhead imposed by WASP on the Web applications that it protects?

TABLE 4
Overhead Measurements for the Macro Benchmarks (RQ3)

<i>Subject</i>	<i># Inputs</i>	<i>Avg Time Uninst (ms)</i>	<i>Avg Ovhd (ms)</i>	<i>% Ovhd</i>
Checkers	1,359	122	5	5%
Office Talk	424	56	1	2%
Empl. Dir.	658	63	3	5%
Bookstore	607	70	4	6%
Events	900	70	1	1%
Classifieds	574	70	3	5%
Portal	1,080	83	16	19%
Daffodil	19	90	6	6%
Filelister	40	172	1	1%
WebGoat	40	940	40	5%





Conclusion

- WASP: Highly automated technique for securing applications against SQL Injection Attacks
 - Positive tainting
 - Accurate and efficient taint propagation
 - Syntax-aware evaluation
 - Minimal deployment requirements
- Future work
 - Use static analysis to optimize dynamic instrumentation
 - Apply general principle to other forms of attacks



Questions

