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

William G.J. Halfond

Alessandro Orso

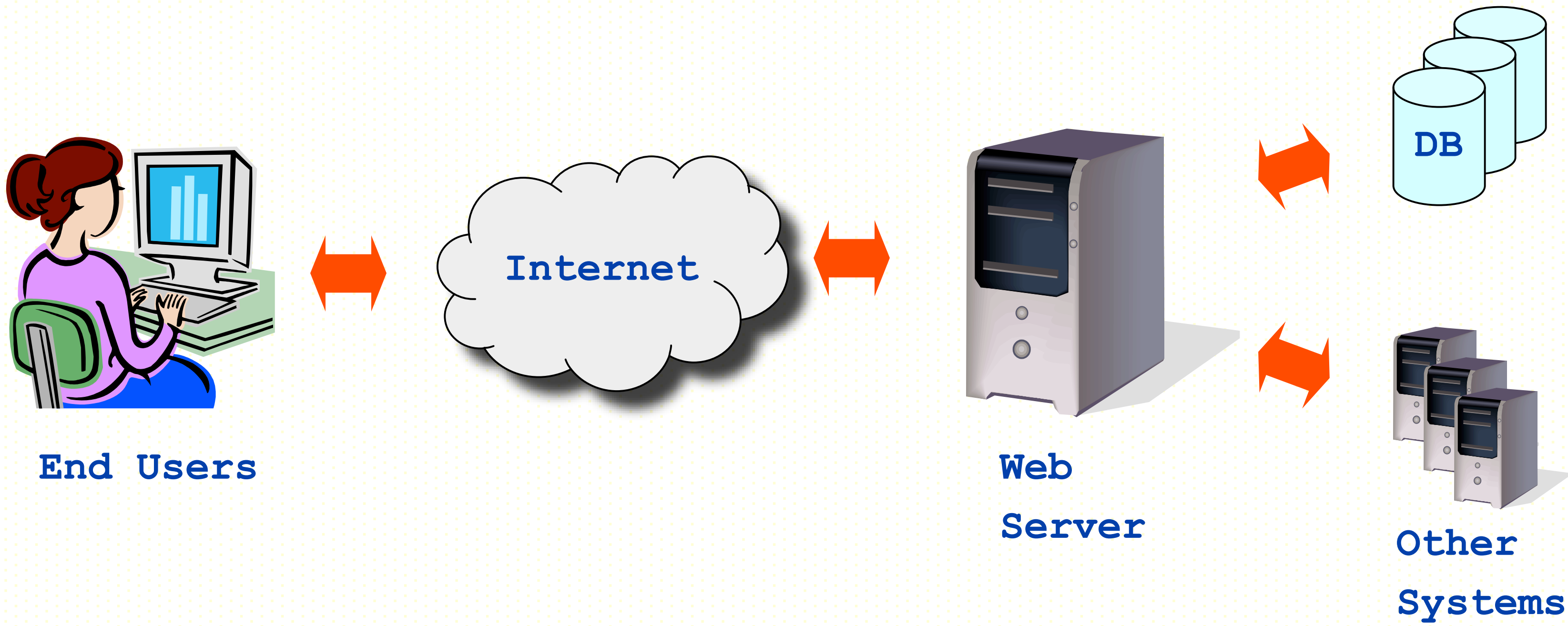
Panagiotis Manolios

Georgia Institute of Technology

Supported by NSF awards CCR-0205422 and CCR-0306372 to GA Tech and by DHS and US Air Force under Contract No. FA8750-05-C-0179.

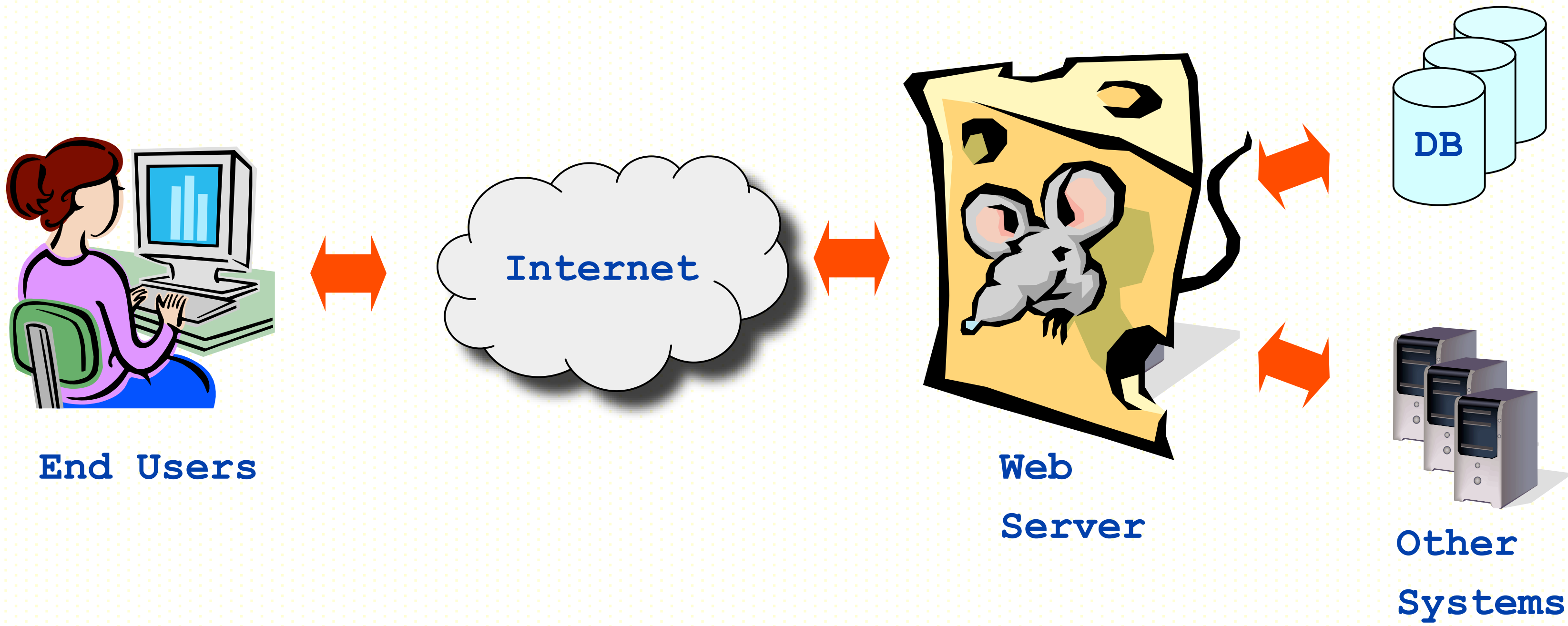
Georgia Tech SPARC Group

Introduction



Deployment context of a typical Web application.

Introduction



Deployment context of a typical Web application.

SQL Injection Attacks

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

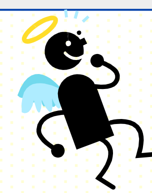
- Open Web Application Security Project (OWASP) lists SQLIA in its top ten most critical web application security vulnerabilities
- David Aucsmith (CTO of Security and Business Unit, Microsoft) defined SQLIA as one of the most serious threats to web apps
- Successful attacks on Guess Inc., Travelocity,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

Example of an SQLIA

```
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

```
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

```
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();
}
```

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

```
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.

Georgia Tech SPARC Group

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] … [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 + "'";
   } else {
4.   queryString+="login='guest'";
   }
5. ResultSet tempSet = stmt.executeQuery(queryString);
```

login -> "doe", password -> "xyz"

queryString
[S][E][L][E][C][T] … [W][H][E][R][E][ ]

tmp0
[l][o][g][i][n][=][']

tmp1
[d][o][e]

tmp2
['][ ][A][N][D][ ][p][a][s][s][=][']

tmp3
[x][y][z]

tmp4
[']

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

… [W][H][E][R][E][ ][l][o][g][i][n][=]['][d][o][e]['][A][N][D][ ][p][a][s][s][=]['][x][y][z][']

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"

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 -> “admin’ -- ”, password -> “”

queryString

… [R][E][][l][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=''

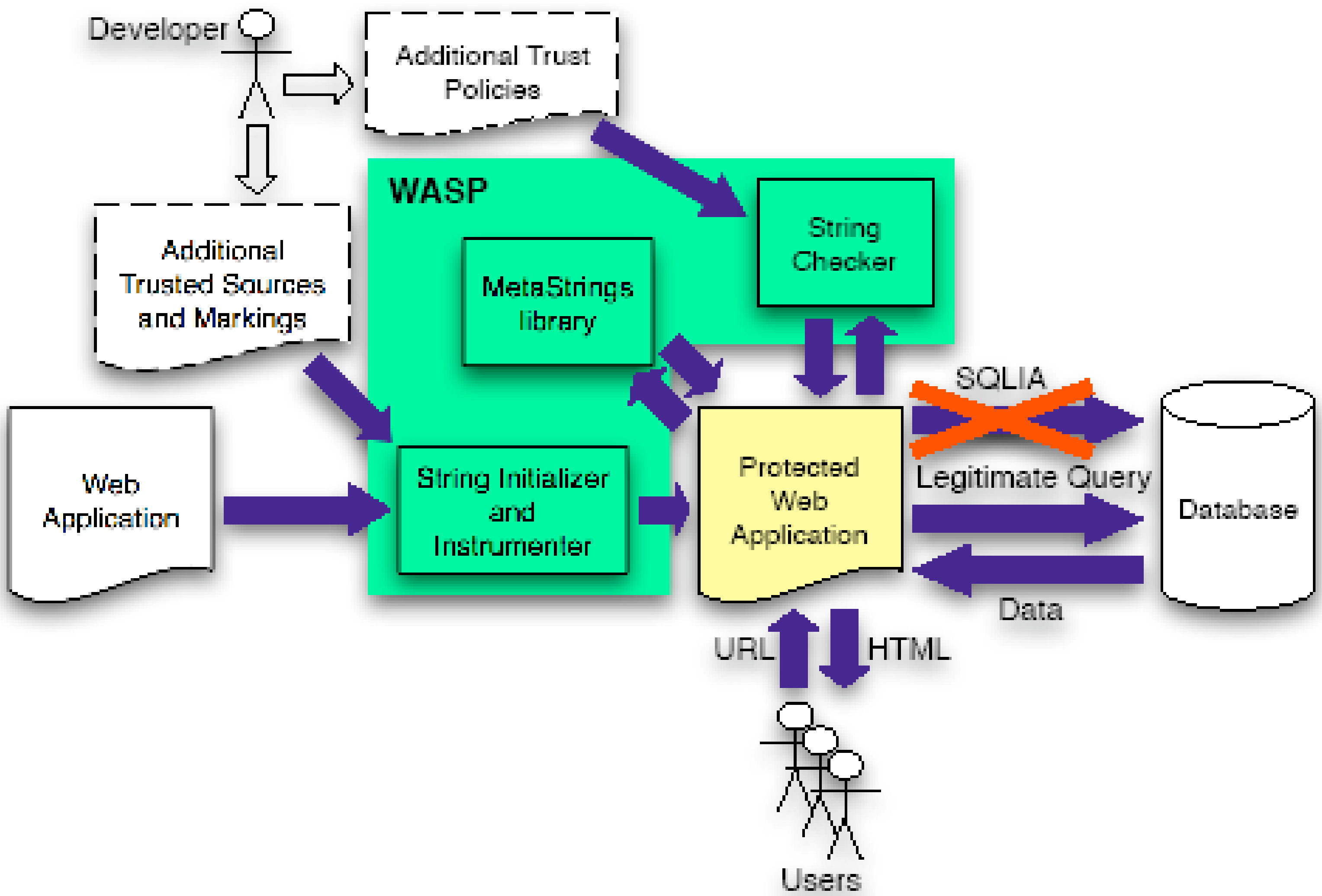
Complete Example

```
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 -> “”

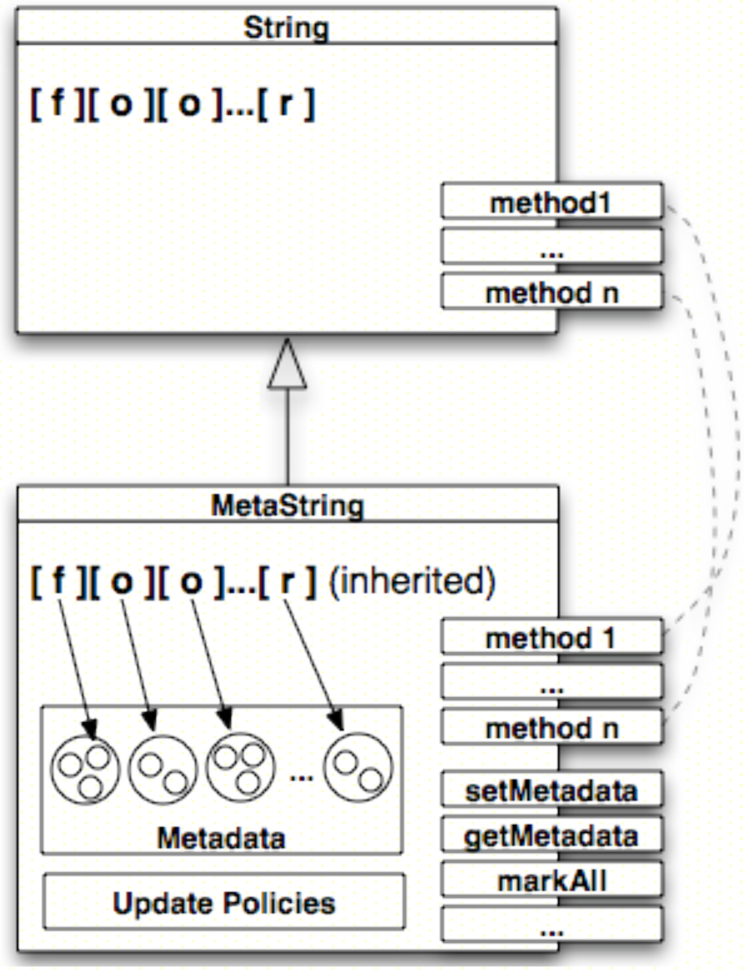
SELECT info FROM userTable WHERE login = ' admin ' -- ' AND pass = ' ' ✘

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

Subject	LOC	Database Interaction Points
Checkers	5,421	5
Office Talk	4,543	40
Employee Directory	5,658	23
Bookstore	16,959	71
Events	7,242	31
Classifieds	10,949	34
Portal	16,453	67

- 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

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

Evaluation Results: Accuracy

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

Evaluation Results: Accuracy

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

No false positives or false negatives in our evaluation.

Evaluation Results: Overhead

Subject	# Inputs	Avg. Access Time (ms)	Avg. Access Overhead (ms)	% Overhead
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%

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?