Preventing SQL Injection Attacks Using AMNESIA

William G.J. Halfond and Alessandro Orso

Georgia Institute of Technology

This work was partially supported by DHS contract FA8750-05-2-0214 and NSF awards CCR-0205422 and CCR-0209322 to Georgia Tech.

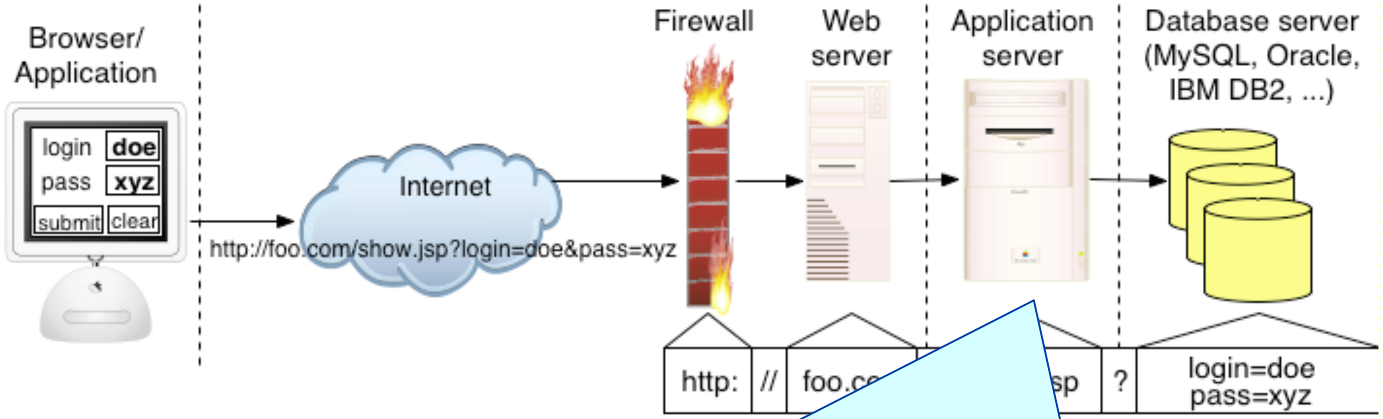
SQL Injection Attacks

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

Presentation Outline

- Motivation
- Background Info.
- AMNESIA
- Demonstration
- Evaluation Overview
- Summary

SQLIA Vulnerability



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

Attack Scenario

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

Normal Usage

¬User submits login “**doe**” and pin “**123**”

  ¬*SELECT info FROM users WHERE login= `**doe**’ AND pin= **123***

Attack Scenario

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

Malicious Usage

- Attacker submits “user’ **--** ” and pin of “0”
  - *SELECT info FROM users WHERE login=‘**user’ --** ’ AND pin=0*

Many types of SQLIA[issse06]

Types

- Piggy-backed Queries
- Tautologies
- Alternate Encodings
- Inference
- Illegal/Logically Incorrect Queries
- Union Query
- Stored Procedures

Sources

- User input
- Cookies
- Server variables
- Second-order
- …

AMNESIA[ase05]

Basic Insights

1. Code contains enough information to accurately model all legitimate queries.
2. A SQL Injection Attack will violate the predicted model.

Solution:

Static analysis => build query models

Runtime analysis => enforce models

Overview of AMNESIA

1. Identify all hotspots.
2. Build SQL query models for each hotspot.
3. Instrument hotspots.
4. Monitor application at runtime.

1 – Identify Hotspots

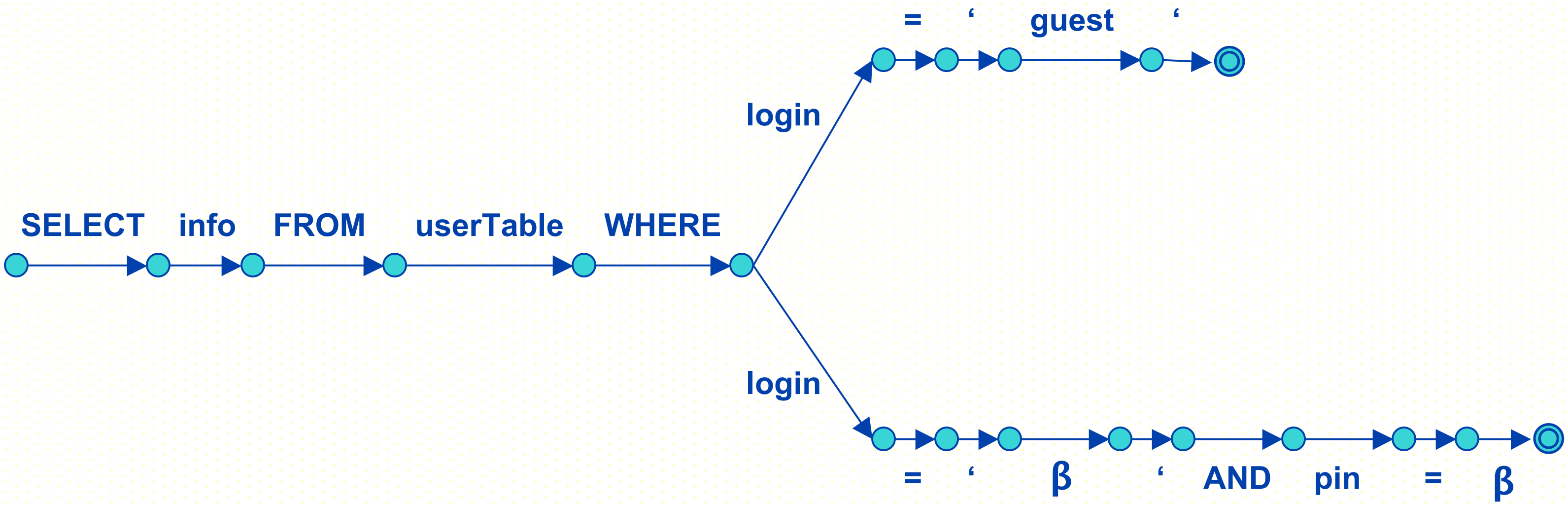
Scan application code to identify hotspots.

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

Hotspot

2 – Build SQL Query Model

1. Use Java String Analysis[1] to construct character-level automata
2. Parse automata to group characters into SQL tokens



3 – Instrument Application

Wrap each hotspot with call to monitor.

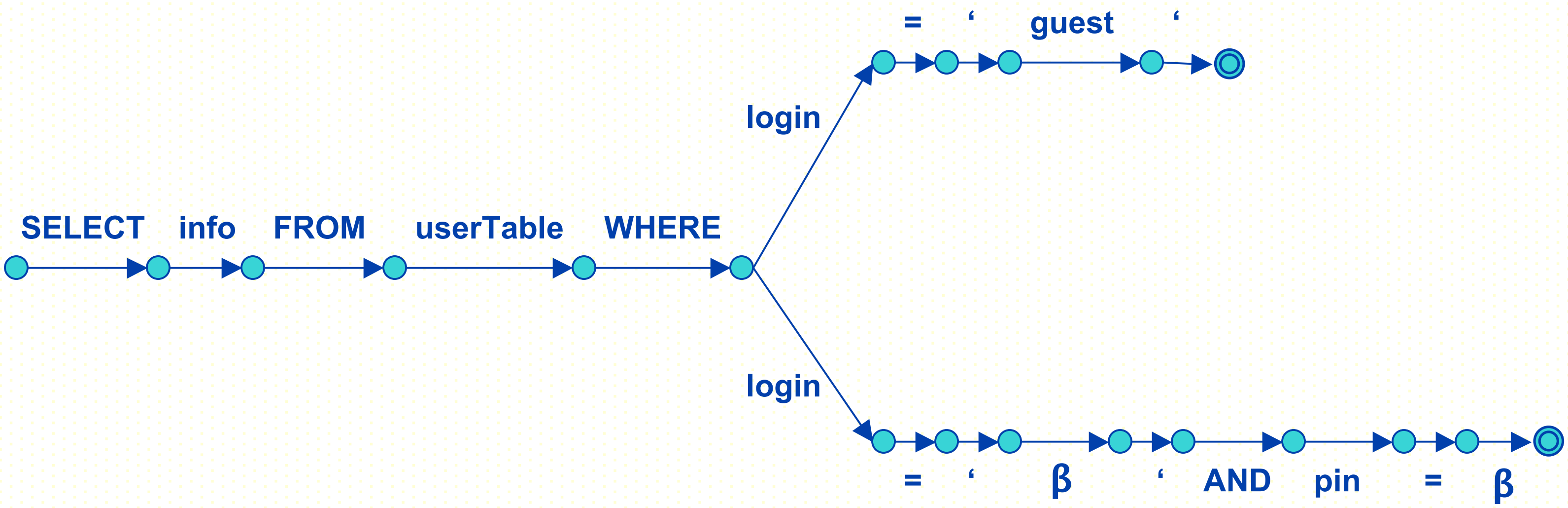
```
String queryString = "SELECT info FROM userTable WHERE ";
if ((! login.equals("")) && (! pin.equals("")))  {
  queryString += "login='" + login + "' AND pin=" + pin ;
} else {
  queryString+="login='guest'";
}

if (monitor.accepts (hotspotID, queryString) {
  ResultSet tempSet = stmt.execute(queryString);
}
```

Call to Monitor

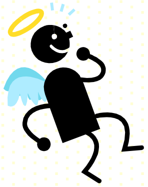
Hotspot

4 – Runtime Monitoring

Check queries against SQL query model.



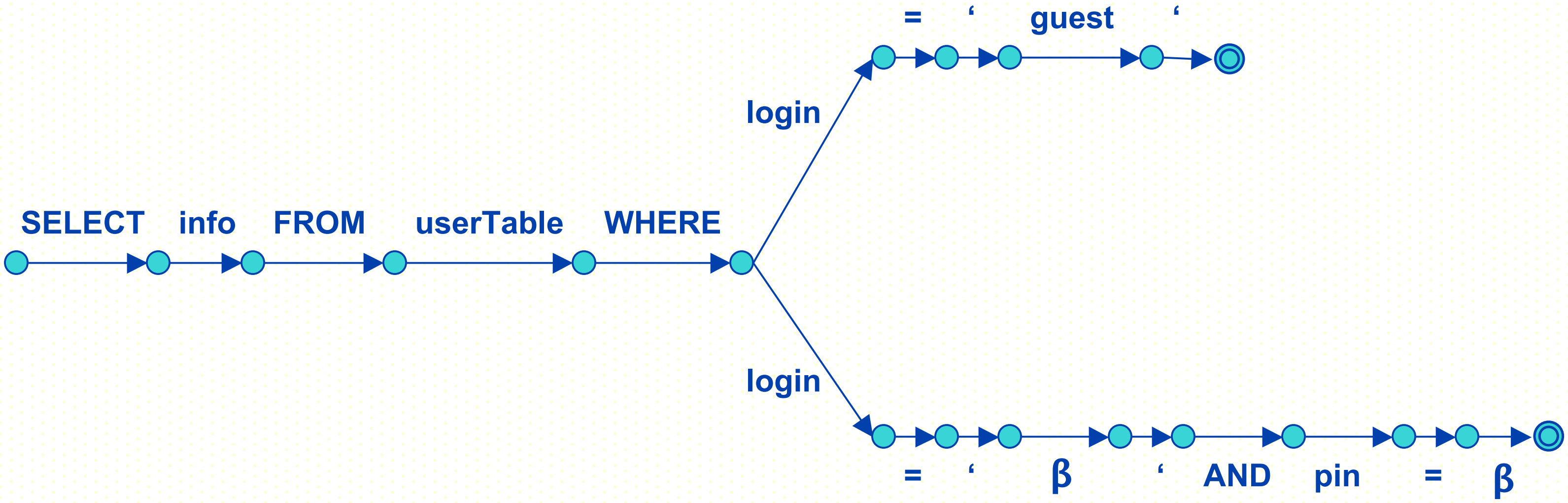
Normal Usage:



SELECT info FROM userTable WHERE login = ‘ doe ‘ AND pin = 123

4 – Runtime Monitoring

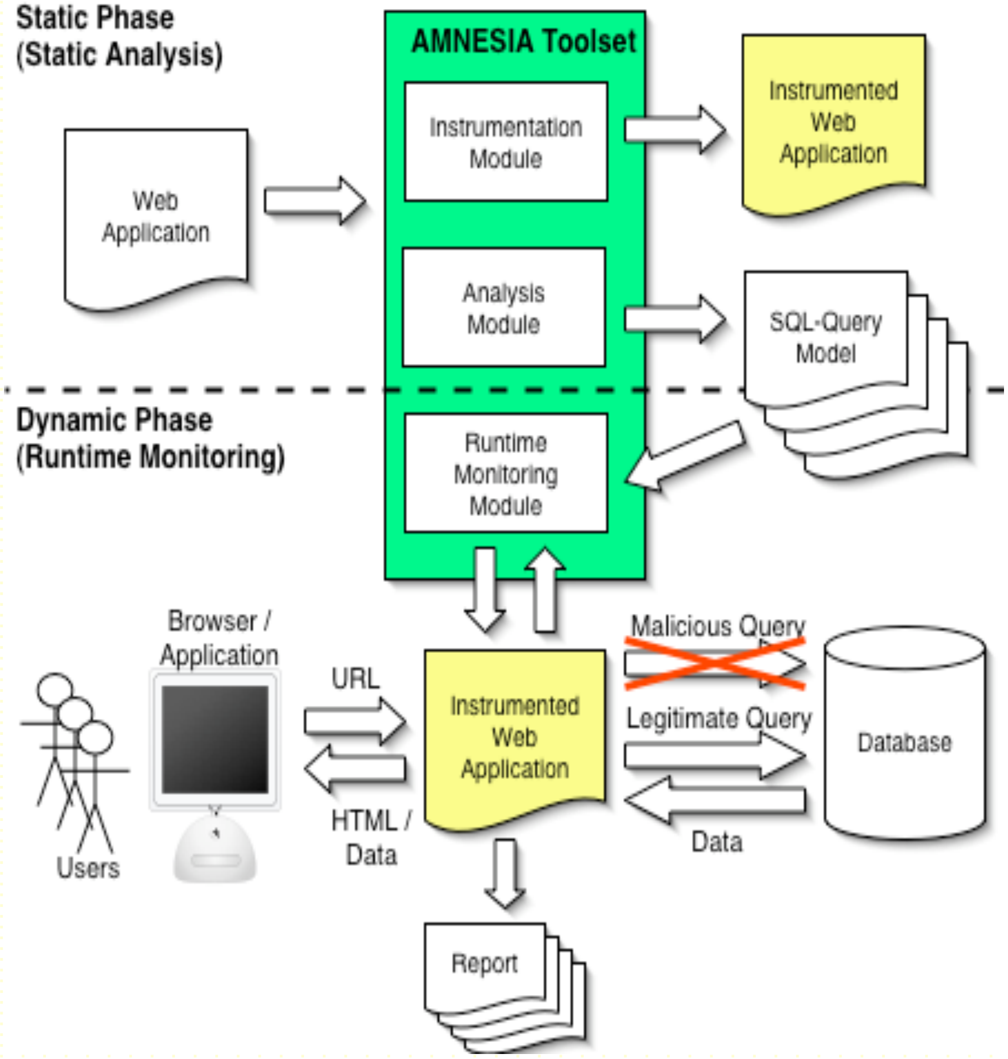
Check queries against SQL query model.



Malicious Usage:

SELECT info FROM userTable WHERE login = ‘ user ‘ -- ‘ AND pin = 0

AMNESIA Implementation



SPARC

AMNESIA Demonstration

- Attacking a commercial application:
  - Evade login protection
  - Change contents of the database – "Special sale price"
- Blocking attacks with AMNESIA
- Examine SQL query models

Evaluation: Research Questions

RQ1: What percentage of attacks can our technique detect and prevent that would otherwise go undetected and reach the database?

RQ2: How much overhead does our technique impose on web applications at runtime?

RQ3: What percentage of legitimate accesses does our technique prevent from reaching the database?

Evaluation: Experiment Setup

Subject	LOC	Hotspots	Average Automata size
Checkers	5,421	5	289 (772)
Office Talk	4,543	40	40 (167)
Employee Directory	5,658	23	107 (952)
Bookstore	16,959	71	159 (5,269)
Events	7,242	31	77 (550)
Classifieds	10,949	34	91 (799)
Portal	16,453	67	117 (1,187)

- 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: RQ1

Subject	Unsuccessful	Successful	Detected
Checkers	1195	248	248 (100%)
Office Talk	598	160	160 (100%)
Employee Directory	413	280	280 (100%)
Bookstore	1028	182	182 (100%)
Events	875	260	260 (100%)
Classifieds	823	200	200 (100%)
Portal	880	140	140 (100%)

⇒ No false negatives

⇒ Unsuccessful attacks = filtered by application

Evaluation Results: RQ2 & RQ3

- Runtime Overhead
  - Less than 1ms.
  - Insignificant compared to cost of network/database access
- No false positives
  - No legitimate input was flagged as SQLIA

Conclusions & Future Work

- AMNESIA detects and prevents SQLIAs by using static analysis and runtime monitoring
  - Builds models of expected legitimate queries
  - At runtime, ensure all generated queries match model
- In our evaluation
  - No false positives
  - No false negatives
- Future work => address limitations
  - Imprecision in static analysis
  - External trusted input