

# Secure Programming Lecture 7: SQL Injection

David Aspinall

Informatics Edinburgh

# Outline

## Overview

Past attacks

Vulnerable code examples

Classification

- Injection route and motive

- Forms of SQL code injected

Prevention and detection

Summary

## Recap

**Injection attacks** use specially crafted inputs to subvert the intended operation of applications.

- ▶ **Operating System Command Injections** may execute arbitrary commands. Most dangerous.
- ▶ **SQL Injections** can reveal database contents, affect the results of queries used for authentication; sometimes they can even execute commands. Often accessible indirectly via web servers; most prolific.

In this lecture we look at **SQL Injections** in some detail.

# Context

SQL Injection (SQLi) has regularly featured high in lists of the most common software vulnerabilities.

- ▶ Akami's [2021 State of the Internet report](#) recorded that SQLi was top of the attack list with 6.2 billion attempts recorded over 18 months.

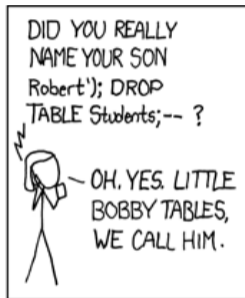
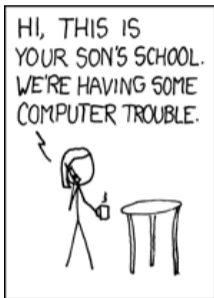
As with overflows, there is a **large body of crafty exploits** made possible by (often small) errors in coding or design.

We will look at:

- ▶ SQLi attack types and mechanisms
- ▶ detecting SQLi
- ▶ preventing SQLi

Even if you believe you are safe from SQLi, it is useful to understand the range of problems and solutions. And a “NoSQL” database doesn't mean no NoSQL injections!

# xkcd #327



## Bobby Tables: A guide to preventing SQL injection

Home

About

ASP

ColdFusion

C#

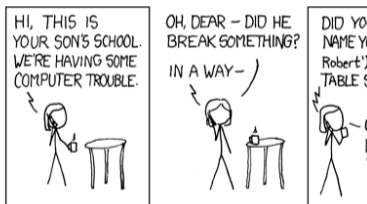
Delphi

.NET

Java

### Who is Bobby Tables?

[From the webcomic xkcd](#)



# SQL Queries

SQL: standard language for interacting with databases

- ▶ very common with **web applications**
  - ▶ authentication: DB of users, passwords
  - ▶ main function often data storage
- ▶ but also in **desktop and server** apps
  - ▶ music player applications, media servers
  - ▶ application caches
  - ▶ custom corporate database clients

**Question.** Why might the second category cause concern for security auditing?

# Network versus local injections

**Network** usually considered the bigger risk

- ▶ Access by many, unknown users
- ▶ Network is gateway, crossing physical boundaries
- ▶ Risk in privileged servers (setuid, etc)

**Local** inputs: should they be considered too?

- ▶ Local users can only deny access to themselves
- ▶ desktop apps run as plain user, only risk own data

However, this trust assumption can be wrong:

- ▶ *drive-by exploits* attack locally (or use escalation)
- ▶ *insider threats* shouldn't be overlooked



# Outline

Overview

**Past attacks**

Vulnerable code examples

Classification

- Injection route and motive

- Forms of SQL code injected

Prevention and detection

Summary

# How I hacked PacketStorm (1988-2000)

-- Advisory RFP2K01 ----- rfp.labs -----

"How I hacked PacketStorm"

A look at hacking wwthreads via SQL

----- rain forest puppy / rfpwiretrip.net ---

- ▶ One of the first public examples and explanation
- ▶ Demonstrated retrieval of 800 passwords
- ▶ See Rain Forest Puppy's [advisory](#) and his earlier [Phrack 54 article](#)

# Man steals 130m card records (2009)

**US prosecutors have charged a man with stealing data relating to 130 million credit and debit cards.**

Officials say it is the biggest case of identity theft in American history.

They say Albert Gonzalez, 28, and two un-named Russian co-conspirators hacked into the payment systems of retailers, including the 7-Eleven chain.

Prosecutors say they aimed to sell the data on. If convicted, Mr Gonzalez faces up to 20 years in jail for wire fraud and five years for conspiracy.

He would also have to pay a fine of \$250,000 (£150,000) for each of the two charges.

## 'Standard' attack

Mr Gonzalez used a technique known as an "SQL injection attack" to access the databases and steal information, the US Department of Justice (DoJ) said.

The method is believed to involve exploiting errors in programming to access data.



The card details were allegedly stolen from three firms, including 7-Eleven

## SQL INJECTION ATTACK

- ✦ This is a fairly common way that fraudsters try to gain access to consumers' card details.
- ✦ They scour the internet for weaknesses in companies' programming which allows them to get behind protection measures.
- ✦ Once they find a weakness, they insert a specially designed code into the network

# Provocation: Swedish pencil vote (2010)

## Did Little Bobby Tables migrate to Sweden?

Posted by Jonas Elfström Thu, 23 Sep 2010 20:36:00 GMT

As you may have heard, we've had a [very close election](#) here in Sweden. Today the Swedish Election Authority published the [hand written votes](#). While scanning through them I happened to notice

```
R;13;Hallands län;80;Halmstad;01;Halmstads västra  
valkrets;0904;Söndrum 4;pwn DROP TABLE VALJ;1
```

The second to last field<sup>1</sup> is the actual text on the ballot<sup>2</sup>. Could it be that [Little Bobby Tables](#) is all grown up and has migrated to Sweden? Well, it's probably just a joke but even so it brings questions since an [SQL-injection](#) on election data would be very serious.

Someone even tried to get some JavaScript in there:

```
R;14;Västra Götalands län;80;Göteborg;03;Göteborg,  
Centrum;0722;Centrum, Övre Johanneberg;(Script  
src=http://hittepa.webs.com/x.txt);1
```

I'm pleased to see that they published the list as text and not HTML. This

# Should know better (2011)



Full Disclosure mailing list archives

By Date

By Thread

Search

## MySQL.com Vulnerable To Blind SQL Injection Vulnerability

---

From: Jack haxor <jackh4xor () h4cky0u org>

Date: Sun, 27 Mar 2011 05:46:30 +0000

---

[+] MySQL.com Vulnerable To Blind SQL Injection vulnerability  
[+] Author: Jackh4xor @ w4ck1ng  
[+] Site: <http://www.jackh4xor.com>

About MySQL.com :

The Mysql website offers database software, services and support for your business, including the Enterprise server, the Network monitoring and advisory services and the production support. The wide range of products include: Mysql clusters, embedded database, drivers for JDBC, ODBC and Net, visual database tools (query browser, migration toolkit) and last but not least the MaxDB- the open source database certified for SAP/R3. The Mysql services are also made available for you. Choose among the Mysql training for database solutions, Mysql certification for the Developers and DBAs, Mysql consulting and support. It makes no difference if you are new in the database technology or a skilled developer of DBA, Mysql proposes services of all sorts for their customers.

Vulnerable Target : <http://mysql.com/customers/view/index.html?id=1170>  
Host IP : 213.136.52.29  
Web Server : Apache/2.2.15 (Fedora)  
Powered-by : PHP/5.2.13  
Injection Type : MySQL Blind  
Current DB : web

Data Bases:

information\_schema  
bk  
certification

# Should know better (2013)



## SECURITY

### Under the microscope: The bug that caught PayPal with its pants down

#### Payment giant suffers textbook SQL injection flaw

By John Leyden, 15th April 2013

42

[The threat landscape](#)

Security researchers have published a more complete rundown of a recently patched SQL injection flaw on PayPal's website.

#### RELATED STORIES

Verizon, Experian and pals bag £25m to inspect Brits' identities for UK gov

The Vulnerability Laboratory research team received a [\\$3,000 reward](#) after discovering a remote SQL injection web vulnerability in the official PayPal GP+ Web Application Service. The critical flaw, which could have been remotely exploitable, allowed hackers to inject commands through the vulnerable web app into the backend databases, potentially tricking them into coughing up sensitive data in the process.

## Should know better (2015)

### **TalkTalk hack: MPs to hold inquiry into cyber-attack**

🕒 26 October 2015 | **Business**

---



## Analysis: Rory Cellan-Jones, BBC technology editor

The company first indicated that the "sustained" attack was a DDoS, a distributed denial of service attack where a website is bombarded with waves of traffic.

That did not seem to explain the loss of data, and later TalkTalk indicated that there had also been what is known as an SQL injection.

This is a technique where hackers gain access to a database by entering instructions in a web form. It is a well known type of attack and there are relatively simple ways of defending against it.

Many security analysts were stunned by the idea that any major company could still be vulnerable to SQL injection.



# Provocation: British company name (2016)

## ; DROP TABLE "COMPANIES";-- LTD

[File for this company](#)

Company number **10542519**

**Overview**

Filing history

People

Registered office address

**1 Moyes Cottages Bentley Hall Road, Capel St. Mary, Ipswich,  
Suffolk, United Kingdom, IP9 2JL**

Company status

**Active**

Company type

**Private limited Company**

Incorporated on

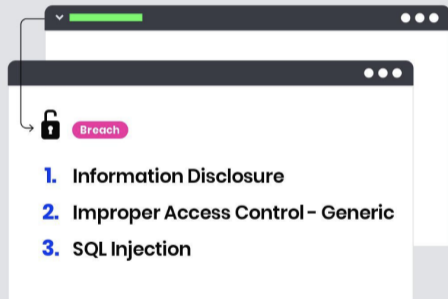
**29 December 2016**

See the [reddit thread](#)

# Bug bounties in Hack U.S. (4th July 2022)

## MOST COMMON VULNERABILITIES

### Top Three Vulnerability Types



"Through initial evaluation of Hack U.S. reporting, the most commonly identified vulnerability is categorized as "Information Disclosure." With the identification of vulnerability trends, we can seek out patterns of detection and ultimately create new processes and system checks to ensure we address the root cause and develop further mitigations against malicious actors who might try to exploit our systems."

#### **Melissa Vice**

VDP DIRECTOR,  
DOD CYBER CRIME CENTER (DC3)

# Outline

Overview

Past attacks

**Vulnerable code examples**

Classification

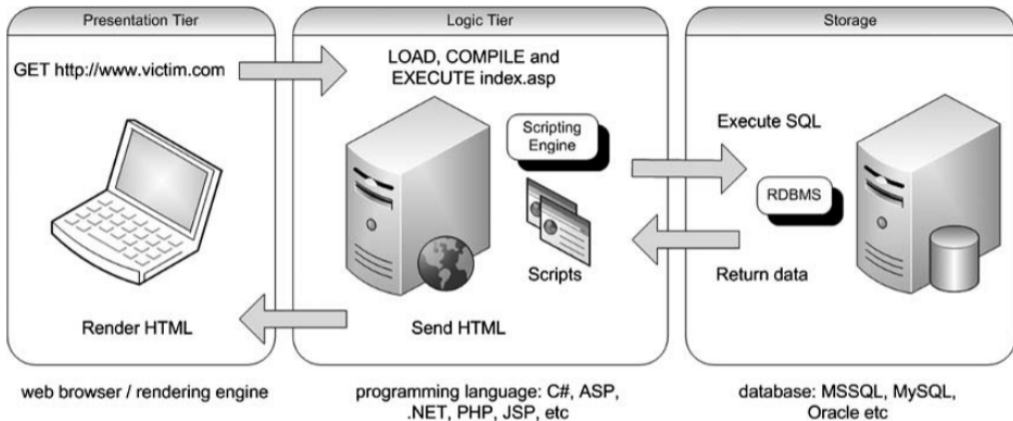
- Injection route and motive

- Forms of SQL code injected

Prevention and detection

Summary

# Typical setting for attacks



Picture from [SQL Injection Attacks and Defense](#), J. Clarke, Syngress, 2012

## Typical vulnerability in PHP code

```
$username = $_HTTP_POST_VARS['username'];  
$password = $_HTTP_POST_VARS['passwd'];  
  
$query = "SELECT * FROM logintable WHERE user = '"  
    . $username . "' AND pass = '" . $password . "'";  
...  
$result = mysql_query($query);  
  
if (!$results)  
    die_bad_login();
```

Guaranteed login! Try with:

*user name:* bob' OR user<>'bob

*password:* foo' OR pass<>'foo

which gives

```
SELECT * FROM logintable WHERE user=  
'bob' or user<>'bob' AND pass='foo' OR pass<>'foo'
```

## Fixes: in-band versus out-of-band

- ▶ The “in-band” solution is to use *filtering* to escape banned characters.

**In-band** use sanitization or filtering to remove banned characters

- ▶ PHP and MySQL, for example, provide functions to help do this, guaranteeing meta-characters are quoted.

**Out-of-band** use a **prepared query** with parameters carved out

- ▶ Parameters get safely substituted into query without interpreting user text as commands

A more general “out-of-band” solution is to use *embedded programming language support*. For example:

- ▶ *Object-Relational Mapping* database is interrogated via objects or
- ▶ *LINQ*, Language-Integrated Query in .NET.

**Question.** Why might out-of-band fixes be preferred?

## An example in Java servlet code

```
1 public class Show extends HttpServlet {
2     public ResultSet getuserInfo(String login, String pin) {
3         Connection conn = DriverManager.getConnection("MyDB");
4         Statement stmt = conn.createStatement();
5         String queryString = "";
6
7         queryString = "SELECT accounts FROM users WHERE ";
8         if ((! login.equals("")) && (! pin.equals(""))) {
9             queryString += "login='" + login +
10                "' AND pin=" + pin;
11         } else {
12             queryString+="login='guest'";
13         }
14
15         ResultSet tempSet = stmt.execute(queryString);
16         return tempSet;
17     }
18 }
```

## Normal usage

```
7   queryString = "SELECT accounts FROM users WHERE ";
8   if ((! login.equals("")) && (! pin.equals(""))) {
9       queryString += "login='" + login +
10          "' AND pin=" + pin;
11   } else {
12       queryString+="login='guest'";
13   }
```

User submits login="john" and pin="1234"

SQL issued:

```
SELECT accounts FROM users WHERE login='john' AND pin=1234
```



# Malicious usage

```
7   queryString = "SELECT info FROM users WHERE ";
8   if ((! login.equals("")) && (! pin.equals(""))) {
9       queryString += "login='" + login +
10          "' AND pin=" + pin;
11   } else {
12       queryString+="login='guest' ";
13   }
```

User submits login="admin' --" and pin="0"

SQL issued:

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

# Quotation and meta-characters

Warnings about meta-characters also apply to SQL. They can vary by DB engine, even by configuration.

MySQL 8.0 Reference Manual / ... / String Literals

version 8.0

## 9.1.1 String Literals

A string is a sequence of bytes or characters, enclosed within either single quote (') or double quote (") characters. Examples:

```
'a string'  
"another string"
```

Quoted strings placed next to each other are concatenated to a single string. The following lines are equivalent:

```
'a string'  
'a' ' ' 'string'
```

If the `ANSI_QUOTES` SQL mode is enabled, string literals can be quoted only within single quotation marks because a string quoted within double quotation marks is interpreted as an identifier.

# Outline

Overview

Past attacks

Vulnerable code examples

**Classification**

**Injection route and motive**

**Forms of SQL code injected**

Prevention and detection

Summary

# Classifying SQL injections

There are a wide variety of SQL injection techniques. Sometimes several are used to mount a single attack.

It's useful to examine:

- ▶ **route** – where injection happens
- ▶ **motive** — what it aims to achieve
- ▶ **SQL code** — the form of SQL injected

These slides follow *A Classification of SQL Injection Attacks and Countermeasures* by Halfond, Viegas and Orso. ISSE 2006.

# Outline

Overview

Past attacks

Vulnerable code examples

**Classification**

**Injection route and motive**

Forms of SQL code injected

Prevention and detection

Summary

# Injection routes

- ▶ **User input** e.g., web forms via HTTP GET or POST
- ▶ **Cookies** used by web apps to build queries
- ▶ **Server variables** logged by web apps (e.g., http headers)
- ▶ In so-called **second-order injections** the injection is separated from attack

# Primary and auxiliary motives

**Primary** motives may be:

- ▶ Extracting data
- ▶ Adding or modifying data
- ▶ Mounting a denial of service attack
- ▶ Bypassing authentication
- ▶ Executing arbitrary commands

**Auxiliary** motives may be

- ▶ Finding injectable parameters
- ▶ Database server finger-printing
- ▶ Finding database schema
- ▶ Escalating privilege at the database level

# Outline

Overview

Past attacks

Vulnerable code examples

**Classification**

Injection route and motive

**Forms of SQL code injected**

Prevention and detection

Summary



## Forms of SQL code injected

1. Tautologies
2. Illegal/incorrect queries
3. Union query
4. Piggy-backed queries
5. Inference pairs
6. Stored procedures and other DB engine features

Additionally, the injection may use *alternate encodings* to try to defeat sanitization routines that don't interpret them (e.g., `char(120)` instead of `x`).

**Exercise.** For each of these types (described next), consider which primary/secondary motive(s) might apply.

# Tautologies

Inject code into condition statement(s) so they always evaluate to true.

```
SELECT accounts FROM users WHERE  
login='' or 1=1 -- AND pin=
```

Blocking tautologies using a deny list is difficult

- ▶ Many ways of writing them:  $1 > 0$ , 'x' LIKE 'x', ...
- ▶ Quasi tautologies: very often true  $RAND() > 0.01$ , ...

**Question.** Instead of a tautology, can you think of how an attacker might use an always-false condition?

# Illegal/incorrect

Cause a run-time error, hoping to learn information from error responses.

```
SELECT accounts FROM users WHERE  
login='' AND pin=convert(int,(select top 1 name from  
sysobjects where xtype='u'))
```

- ▶ Assumes MS SQL server
  - ▶ sysobjects is server table of metadata
- ▶ Tries to find first user table
- ▶ Converts name into integer: runtime error

## Example response

```
Microsoft OLE DB Provider for SQL Server (0x80040E07)  
Error converting nvarchar value 'CreditCards'  
to a column of data type int
```

Tells the attacker:

- ▶ MS SQL Server is running
- ▶ The first user-defined table is called CreditCards

# Union query

Inject a second query using UNION:

```
SELECT accounts FROM users WHERE  
login='' UNION SELECT cardNo from CreditCards where  
acctNo=10032 -- AND pin=
```

- ▶ Suppose there are no tuples with login=' '
- ▶ Result: may reveal cardNo for account 10032

## Piggy-backed (sequenced) queries

The Bobby Tables attack is an example of a piggy-backed query.

The attacker injects a second, distinct query:

```
SELECT accounts FROM users WHERE  
login='doe'; drop table users -- ' AND pin=
```

- ▶ Database parses second command after ';'
- ▶ Executes second query, deleting users table
- ▶ NB: some servers don't need ; character

## Inference pairs

Suppose error responses are correctly captured and *not* seen by the client.

It might still be possible to extract information from the database, by finding some difference between outputs from pairs of queries.

- ▶ A **Blind Injection** tries to reveal information by exploiting some visible difference in outputs.
- ▶ A **Timing Attack** tries to reveal information by making a difference in response time dependent on a boolean (e.g., via `WAITFOR`)

If the attacker has unlimited access, these can be used in repeated, automated, differential analysis.

# Blind injection example

Idea: discover whether login parameter is vulnerable with two tests.

**Step 1.** Always true:

```
login='legalUser' and 1=1 -- '
```

**Step 2.** Always false:

```
login='legalUser' and 1=0 -- '
```



# Blind injection example

## Step 1

```
SELECT accounts FROM users WHERE login='legalUser' and 1=1 -- '
```

### RESPONSE: INVALID PASSWORD

The attacker thinks:

*Perhaps my invalid input was detected and rejected, or perhaps the username query was executed separately from the password check.*

# Blind injection example

## Step 2

```
SELECT accounts FROM users WHERE login='legalUser' and 1=0 -- '
```

### RESPONSE: INVALID USERNAME AND PASSWORD

The attacker thinks:

*Aha, the response is different! Now I can infer that the login parameter is injectable.*

# Stored procedures

**Stored procedures** are custom sub-routines which provide support for additional operations.

- ▶ May be written in scripting languages.
- ▶ Can open up additional vulnerabilities.

```
CREATE PROCEDURE DBO.isAuthenticated
userName varchar2, pin int
AS
EXEC("SELECT accounts FROM users
WHERE login='" +userName+ "' and pass='" +pass+
      "' and pin=" +pin);
GO
```

varchar2 is an Oracle datatype for variable length strings

# Stored procedures

This is invoked with something like:

```
EXEC DB0.isAuthenticated 'david' 'bananas' 1234
```

# Stored procedures

Or something like:

```
EXEC DB0.isAuthenticated ' ' ; SHUTDOWN; -- ' ' ' '
```

which results in:

```
SELECT accounts FROM users WHERE  
login='doe' pass=' ' ; SHUTDOWN; -- AND pin=
```

## An especially dangerous stored procedure

Microsoft SQL Server offers: **xp\_cmdshell**, which allows operating system commands to be executed!

```
EXEC master..xp_cmdshell 'format c:'
```

- ▶ Since SQL Server 2005, this is disabled by default
- ▶ ... but might be switched back on by DB admins
- ▶ ... maybe from inside the db?!

Lesson: **access control and passwords** may be critical inside the DB, even for restricting attacks outside.

## Other database server features

There are other features offered variously depending on the DB engine. For example, queries in MySQL can write files with the idiom:

```
SELECT INTO outfile.txt ...
```

**Question.** Why might writing files be of use to an attacker?

# Outline

Overview

Past attacks

Vulnerable code examples

Classification

- Injection route and motive

- Forms of SQL code injected

**Prevention and detection**

Summary



# How do I repair an SQLi vulnerability?

Mentioned earlier:

- ▶ *filtering* to sanitize inputs
- ▶ *prepared* (aka *parameterized*) queries

Both methods are server side, so it is better to use database driver  
libraries to abstract away from the underlying DB engine.

In Java, JDBC provides the PreparedStatement class.

We'll look at further relevant secure coding issues later lectures; in particular, ways of  
managing input and also *output* filtering.

**Question.** What type of SQL attacks might PreparedStatements not  
prevent against?

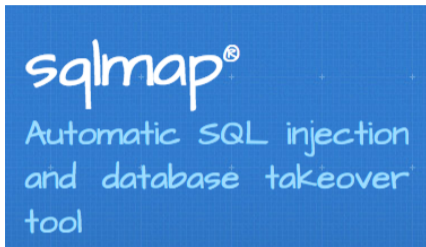
# How do I prevent SQLi vulnerabilities?

Choice of stages (as usual):

1. eliminate before deployment:
  - ▶ use programming language support; object-relational mapping
  - ▶ manual code review or automatic static analysis
2. in testing or deployment:
  - ▶ pen testing tools
  - ▶ instrumented code
3. after deployment:
  - ▶ wait until attacked, manually investigate
  - ▶ use dynamic remediation plus alarms (app firewall or specialised technique)

Some examples follow.

## Detection externally: pen testing tools



Tools like these incorporate the injection methods shown before, to explore a server for typical vulnerabilities.

# Static prevention: automated analysis

Idea: static code analysis used to warn programmer or prohibit/fix vulnerable code.

Techniques:

- ▶ Detect suspicious code patterns, e.g., dynamic query construction
- ▶ Use static taint analysis to detect data-flows from input parameters to queries

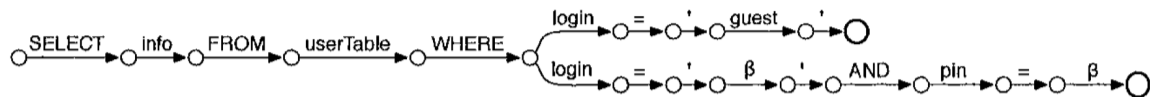
We'll look at static analysis generally in more detail later.

## Dynamic detection tool: AMNESIA

Idea: use static analysis pre-processing to generate a *dynamic* detection tool:

1. Find SQL query-generation points in code
2. Build *SQL-query model* as NDFA which models SQL grammar, transition labels are tokens
3. Instrument application to call runtime monitor
4. If monitor detects violation of state machine, triggers error, preventing SQL query

# State machine for SQL production

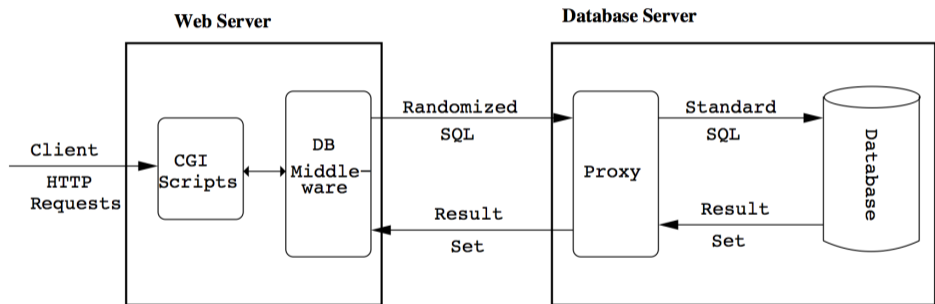


- ▶ Variable  $\beta$ : matches any string *in SQL grammar*
- ▶ Spots violation in injectable parameters
  - ▶ abort query if model not in accepting state

See Halfond and Orso, [AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks](#), Automated Software Engineering, 2005

# Dynamic prevention: SQLrand

Idea: use *instruction set randomization* to change language dynamically to use opcodes/keywords that attacker can't easily guess.



See Boyd and Keromytis, [SQLrand: Preventing SQL Injection Attacks](#), Applied Cryptography and Network Security, 2004

# Outline

Overview

Past attacks

Vulnerable code examples

Classification

- Injection route and motive

- Forms of SQL code injected

Prevention and detection

**Summary**



# Review questions

## SQLi classification

- ▶ Describe three routes for SQL injection.
- ▶ Describe three auxiliary motives that an attacker may have when using SQL injection techniques to learn about a target.

## SQLi prevention and detection

- ▶ How would you repair the prototypical example SQLi vulnerability?
- ▶ Describe automatic ways to *prevent* and *detect* SQLi vulnerabilities.

## References and credits

This lecture includes content adapted from:

- ▶ *A Classification of SQL Injection Attacks and Countermeasures* by Halfond, Viegas and Orso. ISSE 2006
- ▶ *SQL Injection Attacks and Defense*, Edited by Justin Clarke, Syngress. 2nd Edition 2012.