Mitigation of SQL Injection Risks Through Web Application Frameworks

Prepared by Christina Paradis

1. Introduction

Database-backed web applications that accept and execute any syntactic content [such as Standard Query Language (SQL) commands] included with user input are vulnerable to SQL injection attacks. In recent years, several high-profile websites have been compromised by SQL injection attacks: Intel in December 2009[10], the British Royal Navy in November 2010[9], and the US Army in January 2010[8]. The 2010 Midyear Trend and Risk Report from IBM X-Force [59] states: "Web application vulnerabilities—particularly cross-site scripting and SQL injection—continue to dominate the threat landscape." The report also states that SQL injection attacks "continue to be creatively concealed to bypass many security products."

The web applications that provide access to databases are attractive targets for attacks because the interfaces are publicly accessible over the internet and the back-end databases they connect to may contain valuable information. They are vulnerable to attack because the most straight-forward way for a programmer to build a database-backed web application is to construct SQL command statements from quoted strings unrecognized by the compiler. These queries are finalized only at runtime, rely on input from users, and are executed by the database using the privileges granted to the application rather than individual web users.

When Yahoo, the first web search engine, was launched in 1995, most web pages consisted of text, hyperlinks, and JPEG or GIF images. By early 1999, database-backed internet applications had caught the attention of MIT professors Hal Abelson, Michael Dertouzos, and Philip Greenspun, who developed MIT course 6.916: Software Engineering of Innovative Web Services. In the course, students learned to design a data model, specify a page flow, and implement the system using SQL and a procedural language to wrap the results of an SQL query in an HTML template. Around the same time, an article appeared in *Phrack* magazine on Christmas Day 1998[41], which is believed to have been the first publicized mention of SQL injection.

Since then, attackers have leveraged automated tools to exploit hundreds of thousands of vulnerable websites. In May 2008, the average volume of SQL injection attacks was close to 5,000. During the summer of 2009, IBM X-Force began seeing 600,000 SQL injection events per day. Tom Cross, manager of IBM X-Force Research said in an interview in 2009 [40] that attackers had begun using automated tools to use SQL injection to redirect and exploit traffic from legitimate websites.

During an SQL injection attack, a malicious user attempts to change the intended effect of a query command that is generated dynamically by the web application by inserting syntactic content into the query.

Here is an example of an SQL injection attack:

a) User input is collected via a form on a web page (as shown in Figure 1) containing a textbox called `webform_lname`.

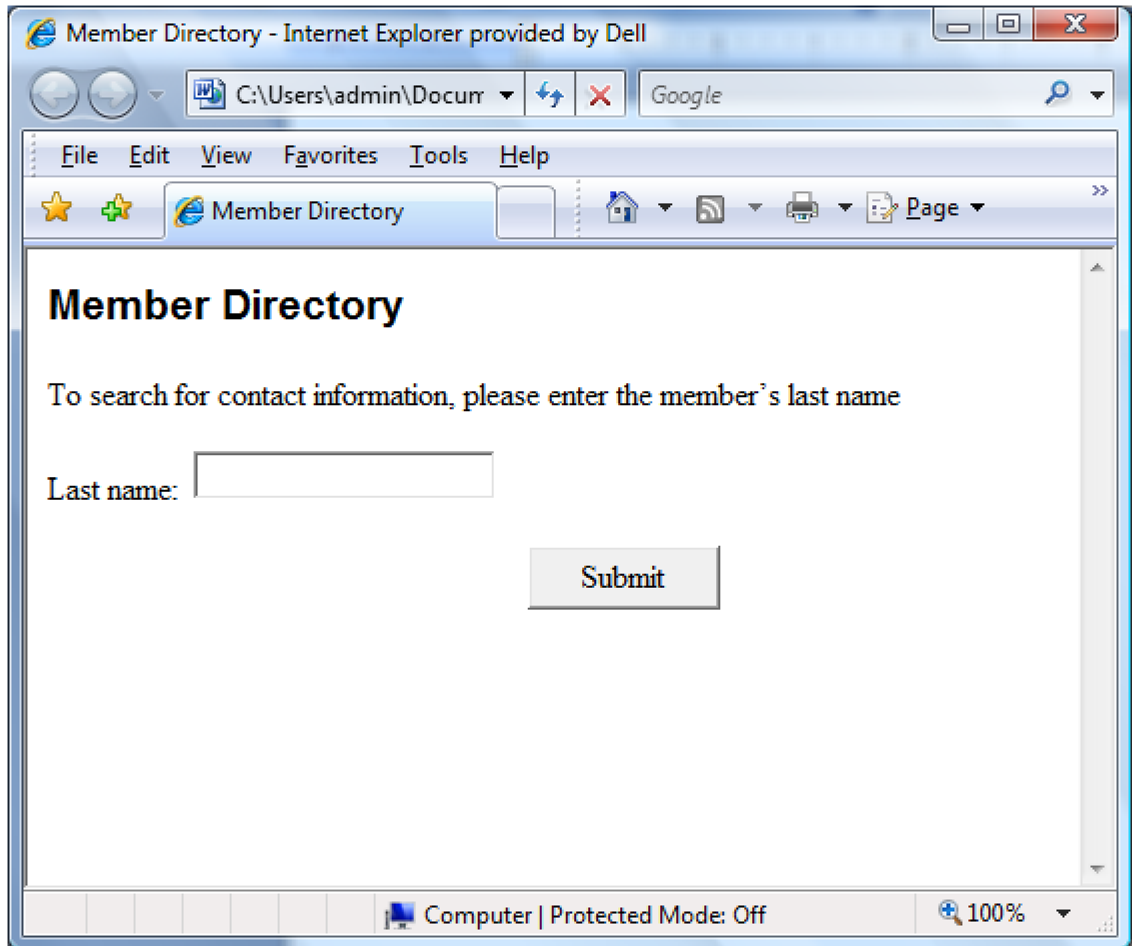

Figure 1 — A Web Form

b) When the submit button is clicked, the web application code builds the SQL command:

```
$query = "SELECT fname, phone, fax, email
FROM members
WHERE lname = '$webform_lname';"
```

c) Instead of giving the expected value of a last name of a member of the organization, a malicious user enters the following into the last name field:

```
'; SELECT password, phone, fax email FROM members
WHERE lname= 'Smith'
```

d) The query that the web application will create and execute on the database becomes:

```
SELECT fname, phone, fax, email FROM members
WHERE lname = '';
SELECT password, phone, fax email FROM members
WHERE lname = 'Smith';
```

e) The new query will return results that include Smith's password:

```
Golfstar1 555-1000 555-1001 smith@bigcompany.com
```

Note that in the SQL injection attack example above, the new query includes syntactic content (executable code) that changes the structure of the query from what the programmer originally intended, and returns data that should have been protected.

Web applications that accept and execute any syntactic content included with user input are vulnerable to injection attacks. The best place to ensure that syntactic content is successfully prevented from being executed is at the point where the data access services are defined. When an application architecture includes a web application framework, these data access services are usually provided by the framework. Software development practitioners such as architects, application designers and programmers rely on the framework to provide services that create safe queries.

The goal of this project is to examine whether commonly used web application frameworks can be an effective way to create web applications that are free from SQL injection vulnerabilities. Each framework presents a learning curve to master the database access API (Application Programming Interface). Some frameworks offer multiple methods to access the database, including ways for insufficiently trained developers to create queries that introduce SQL injection vulnerabilities into the application. Through careful analysis of these database access methods, this project compares the different web application frameworks and points out the need for caution in the use of particular methods.

2. Background

In order to fully understand the problem of SQL Injection vulnerabilities, it is important to understand that the problem arises because of the heterogeneous nature of a database-backed web application. The application programming language must perform application logic, translate and present commands to the database in SQL, parse the query results, and finally wrap the information in HTML (Hyper Text Markup Language) tags to present to the web browser. In order to be interactive, the application must also accept user input. If the user input includes malicious commands, the application must not execute them.

This section describes the different kinds of user input (semantic and syntactic), the architecture that organizes the way the application interacts with the database and the web server, and the details of the way the queries are constructed and the database connections are made.

2.1 Semantic vs. Syntactic Data

SQL injection attack methods exploit dynamically-generated queries that rely on user input. Web applications need to be able to accept semantic input from users in order to provide an interactive user experience. Semantic data includes search terms, authentication credentials such as username and password, and literal values to be stored in database tables. It does NOT include executable commands. User input that is entirely comprised of semantic data poses no injection threat.

In addition to executing queries based on semantic input, the web application must prevent syntactic data input by users from being executed. Syntactic content is executable code. It includes:

a) SQL keywords

b) SQL operators such as special characters that denote a literal search term (‘’), a comment (--) or the end of a command line (;)

c) calls to stored procedures

d) other commands that may be executed by the database software or at the command line of the operating system on which the database resides.

In the previous attack example the expected user input is an example of semantic data, the last name of a member of the organization. The malicious input included several examples of syntactic data, including the SQL keywords “SELECT” and “FROM” and “WHERE”, a table name “members,” and two column names “password” and “phone,” and the SQL operators ‘ and ;

```
’; SELECT password, phone, fax email FROM members
WHERE lname= ‘Smith’
```

2.2 Alternate Encodings

One obvious way to approach the problem of separating syntactic content from user input is to add transformational routines to the web application code to strip away SQL keywords such as OR and the ‘ symbol. However, in addition to the false positives raised by legitimate semantic content (such as the names Dora or O’Brian), the SQL language contains many ways to express a statement. For example, the = sign may be replaced by the expression eq.

Another even greater concern is the existence of many character encoding schemes such as hexadecimal, ASCII or Unicode and many others. In a character encoding scheme each character has a numeric equivalent; for example the ASCII code for the character A is 65. In addition to filtering for SQL commands and operators in clear text, the application must also recognize their character equivalent in any encoding system that would be recognized by the system.

Attackers use these alternate encoding methods to obfuscate the attacks, making them difficult to detect and defend against. The technique of obfuscation is not limited to SQL injection vulnerabilities, but is also commonly used to hide malicious JavaScript. Obfuscation is considered a rising trend in vulnerabilities and exploitation[59].

2.3 Architecture of a Multi-tiered Web Application

Web applications are often created using a multi-tiered architecture. Tiers refer to the physical distribution of components of a system on separate servers such as a Web Server (for example, Apache or IIS), an Application Server or Framework (for example, .NET, Ruby on Rails) and a Database Server (for example, Oracle or SQL Server). These separate tiers allow the application to access the data contained in the database and present it to the user via the web server as html pages.

Figure 2 presents a logical, rather than a physical, view of the architecture. The logical groupings of components are called layers and may not directly correlate to physical servers. The Client Layer, Application Layer, Data Access Layer and Data Layer are shown. Physically, the Data Access Layer is

part of the web application framework. Logically, it insulates the web application logic from the technical specifics of how the information is stored and accessed.

In the web application architecture shown in Figure 2, the user input is collected in the client layer and the database queries are created in the data access layer. The best place to ensure that syntactic content is successfully prevented from being executed is at the data access layer. The mere presence of a data access layer does not prevent SQL injection vulnerabilities, but it does provide a single method of accessing the database. By centralizing this single method of access, rather than having database connections intermixed throughout the codebase, it makes the data access method standardized. If the data access method is safe from SQL injection vulnerabilities at this one centralized point, it will be secure throughout the application. This makes the code easier to read and review for security risks.

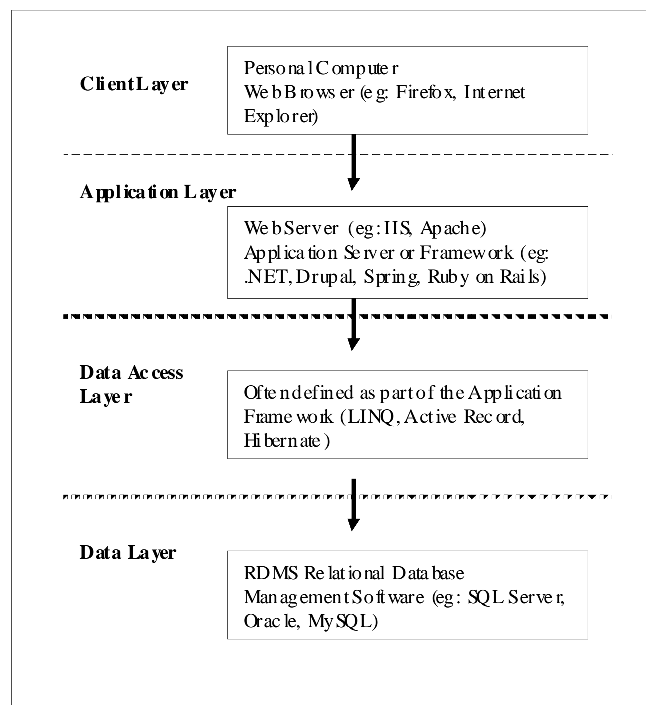

Figure 2— Architecture of a Multi-Tiered Web Application

A web application framework such as Drupal, Ruby on Rails, or Django is not strictly required. There are many multi-tiered web applications that were created without using any web application framework at all. These applications contain all of their functionality within the application code.

The main advantage of using a web application framework is that the most commonly used functionalities are pre-built and available as soon as the framework is installed. Developers need to build only the non-standard, highly customized parts of their application. This makes it easier and faster to build fully functioning database-backed web applications.

Some characteristic functionalities of a web application framework include security services such as authentication and authorization, URL mapping to interpret URLs and direct the user to the correct page, a page template system to display stored content, and a caching system to improve performance. Many frameworks provide a secure method of constructing queries with user input, while also allowing developers to bypass the data access layer entirely and embed SQL queries directly into the code.

Web application frameworks can be so easy to use that they may enable novice programmers with limited understanding of database connections to build sophisticated applications quickly. As a result, they may give developers a false sense of security with regards to SQL injection vulnerabilities. A novice developer may bypass the data access layer without understanding the need to protect the custom embedded SQL queries from malicious user input.

2.4 Parameterized Queries

One important aspect to consider is whether the queries are constructed as parameterized queries, which are sometimes referred to as prepared statements. In a parameterized query, inputs (parameters) are supplied when the query is constructed. Parameters are treated as a literal at runtime and thus will not include syntactic content from the user input.

Schmitt[46] offers this example of an unsafe .NET query:

```
protected void btnSearch_Click(object sender, EventArgs e)
{
String cmd = "SELECT [CustomerID], [CompanyName], [ContactName]
FROM [Customers] WHERE CompanyName ='" + txtCompanyName.Text
+ "'";
SqlDataSource1.SelectCommand = cmd;
GridView1.Visible = true;
}
```

Here is the same code written (safely) with a parameterized query:

```
protected void btnSearch_Click(object sender, EventArgs e)
{
String cmd = "SELECT [CustomerID], [CompanyName], [ContactName]
FROM [Customers]
WHERE CompanyName = @CompanyName";
SqlDataSource1.SelectCommand = cmd;
GridView1.Visible = true;
}
```

OWASP (The Open Web Application Security Project) lists parameterized queries as "Defense Option 1" in its SQL Injection Prevention Cheat Sheet, because using parameterized queries "ensures that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker [58]."

2.5 Database Connections

Another aspect to consider is whether it is possible for programmers to connect to the database directly rather than through the data access layer. The data access layer's main purpose is to help protect developers from security vulnerabilities and centralize functions to avoid repetitive and non-standard code. Functions of the data access layer as shown in Figure 2 typically include following;

a) Map data between the business application and the physical data structure

b) Manage lookup methods and Create, Read, Update and Delete operations with the data store

c) Handle data-type conversions

d) Handle data access/retrieval related errors.

Many frameworks allow programmers to connect to the database directly by to creating custom SQL commands that bypass the data access layer. This flexibility may result in SQL injection vulnerabilities, and should be considered carefully before being implemented. Such custom code requires thorough testing.

3. Methods of SQL Injection Attack

Many types of SQL injection attack methods exist. Researchers such as Halfond, Viegas and Orso [17], and other experts such as Joshi [20], Kost [23] and Maor [27] have classified and described the types of SQL injection attacks in journal articles and white papers.

The different methods of SQL injection attacks include:

a) Tautologies

b) Illegal/Logically Incorrect Queries

c) Union Query

d) Piggy-Backed Queries

e) Stored Procedures

f) Inference

g) Alternate Encodings

h) Second-order injection attacks

3.1 Tautologies

Tautologies are logical statements that are inherently true; for example, a value is always equal to itself. They are useful to a malicious attacker because, when used with the OR keyword, a tautology can alter the test condition intended by the developer so that it returns true for every row in the table [3] [5] [6] [13] [17] [20] [23] [26] [49] [56].

Here is an example of an SQL injection attack that uses a tautology:

a) The web application code builds the SQL command

```
$query = "SELECT fname, phone, fax, email
FROM members
WHERE lname = '$webform_lname';";
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
'or 1=1
```

c) The query that the web application will create and execute on the database becomes:

```
SELECT fname, phone, fax, email FROM members
WHERE lname = '' or 1=1;
```

d) The new query will display all the rows in the `members` table, because the WHERE clause would return true for every row, since 1 always equals 1.

```
Scott 555-1000 555-1001 smith@bigcompany.com
Linda 423-2837 423-2838 jones@bigcompany.com
Paul 423-2837 423-2838 miller@bigcompany.com
```

…

One obvious way to prevent this would be to add transformational routines to the web application code to strip away the OR keyword or the '=' character. However, because SQL is a rich language, there are many ways to express a statement. To subvert this simple fix, a malicious user may craft a command that makes use of "eq" instead of '=' or use ASCII character equivalents, or use a statement such as 2 > 1. The process of sanitizing input is not trivial.

3.2 Illegal/Logically Incorrect Queries

By provoking various database errors an attacker can discover useful information such as the type and version of the backend database. In addition, by extending this technique, an attacker can discover the table names and column names and types of each column contained within the databases [3] [4] [13] [17] [20] [25] [29].

Here is an example of an SQL injection attack that uses illegal/logically incorrect queries:

a) The web application code builds the SQL command

```
$query = "SELECT fname, phone, fax, email
FROM members
WHERE lname = '$webform_lname';";
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
convert (int,(select top 1 name from sysobjects where
xtype='u'))
```

c) The query that the web application will create and execute on the database becomes:

```
SELECT fname, phone, fax, email FROM members
WHERE lname = 'convert (int,(select top 1 name from
sysobjects where xtype='u'))';
```

d) This will cause a database error. Here is an example of an error from a Microsoft SQL Server database server:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'

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

e) There are two key pieces of information returned in the body of the error message:

1) The back-end database is a Microsoft SQL Server and,

2) The name of the first user-defined table in the database is "members". The query could be modified slightly to return the names of the second, then the third user-defined table.

These types of attacks can be ameliorated somewhat by locking down the database server in production so that it does not display errors. While errors are quite useful during development, they are clearly dangerous on a live server. However, inference attacks (described below) can yield similar results without the benefit of the information displayed in the error messages.

3.3 Union Query

A malicious user may attempt to access the database using the UNION SQL keyword, which allows multiple queries to be joined and returns a single result set [4] [6] [13] [17] [20] [23] [27] [29] [35] [36] [44] [47] [57].

Here is an example of an SQL injection attack that uses a union query:

a) The web application code builds the SQL command

```
$query = "SELECT fname, phone, fax, email
FROM members
WHERE lname = '$webform_lname';";
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
'; UNION SELECT visacardno FROM payments
```

c) The query that the web application will create and execute on the database becomes:

```
SELECT fname, phone, fax, email FROM members
WHERE lname = '';
UNION SELECT visacardno FROM payments;
```

d) The new query will display the information stored in the 'visacardno' column for all the rows in the "payments" table. When the results are not restricted by a "where" clause, all results are returned.

```
1234123412341234
3456345634563456
4567456745674567
…
```

One obvious way to prevent this type of attack would be to add transformational routines to the web application code to strip away the UNION keyword from any user input, as well as ASCII and hex equivalents. As mentioned above, it is difficult to ensure that the user input is sanitized enough to prevent all of the possible malicious equivalent strings.

3.4 Piggy-Backed Queries

A piggy-backed query attack vulnerability is present when the application allows multiple queries to be sent to the database in one command string [13] [17] [20] [23]. The attacker may use a semicolon to end one line of a SQL command and begin another.

a) The web application code builds the SQL command

```
$query = "SELECT fname, phone, fax, email
FROM members
WHERE lname = '$webform_lname';";
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
'; DROP TABLE members
```

c) The query that the web application will create and execute on the database becomes:

```
SELECT fname, phone, fax, email FROM members
WHERE lname = '';
DROP TABLE members
```

d) The new query will return no results, but if successful the entire table "members" will be deleted.

Defending against this kind of attack poses the same challenges as sanitizing user input.

3.5 Stored Procedures

A stored procedure is a pre-compiled set of SQL statements with an assigned name that is stored on a database server. It can be run by a user or an application. Certain stored procedures or packages can be powerful tools for a malicious attacker [3] [13] [17] [20] [23] [57]. In the example below, the attacker is able to shut down the database server completely:

a) The web application code builds the SQL command

```
$query = "SELECT fname, phone, fax, email
FROM members
WHERE lname = '$webform_lname';";
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
'; UNION xp_cmdshell(shutdown)
```

c) The query that the web application will create and execute on the database becomes:

```
SELECT fname, phone, fax, email FROM members
WHERE lname = '';
UNION xp_cmdshell(shutdown)
```

d) The query will return no results, but the operating system of the database host computer will shut down, because of the stored procedure `xp_cmdshell`, which allows string commands to pass directly to the operating system at the command line, executed with the same privilege level as the database process. Once the database host computer is shut down, the web application will cease to function until the database server is rebooted.

One way to ameliorate this risk is to lock down the production server by removing any dangerous pre-installed stored procedures that are not being used. However, this will not help if there are stored procedures that are required by applications, or custom stored procedures that are vulnerable.

OWASP[58] lists significant or exclusive use of Stored Procedures as "Defense Option 2," with the lukewarm caveat that "it is possible, although relatively rare, to create a dynamic query inside of a stored procedure that is subject to SQL injection." However Halfond[17], states "It is a common misconception that using stored procedures to write Web applications renders them invulnerable to SQLIA."

Wei[57] offers this example of a vulnerable stored procedure:

```
1. CREATE PROCEDURE [EMP].[RetrieveProfile] @Name varchar(50),
@Passwd varchar(50)
2. WITH EXECUTE AS CALLER
3. AS
4. BEGIN
5. DECLARE @SQL varchar(200);
6. ...
7. SET @SQL='select PROFILE from EMPLOYEE where ';
8. ...
9. IF LEN(@Name) > 0 AND LEN(@Passwd) > 0
10. BEGIN
11. ...
12. SELECT @SQL=@SQL+'NAME="'+@Name+"' and ';
13. SELECT @SQL=@SQL+'PASSWD="'+@Passwd+"";
```

```
14. ...
15. END
16. ELSE
17. BEGIN
18. ...
19. SELECT @SQL=@SQL+'NAME="Guest"';
20. ...
21. END
22. ...
23. EXEC(@SQL)
24. ...
25. END
```

Here is an example of an SQL injection attack against the stored procedure above:

a) When user inputs are provided for @Name and @Passwd, the following query would get executed:
```
select PROFILE from EMPLOYEE where NAME='name' and
PASSWD='passwd'
```
b) Instead of giving the expected value for @Name, a malicious user enters the following:
```
'or 1=1 --
```
c) The query that the web application will create and execute on the database becomes:
```
select PROFILE from EMPLOYEE where NAME='' or 1=1 --and
PASSWD = ''
```
d) The new query will display all the rows in the EMPLOYEE table, because the WHERE clause would return true for every row, since 1 always equals 1.

Web applications that rely on stored procedures can be vulnerable to the same range of attacks as traditional application code. Any dynamically created query that relies on user input should be constructed using parameters that are treated as literals so that all syntactic content of the input is ignored.

3.6 Inference

There are techniques that attackers use to circumvent a properly locked-down database server that is not providing information-rich error messages [3] [17] [27] [48].

Here is an example of an SQL injection attack that uses inference techniques:

a) The web application code builds the SQL command:
```
$query = "SELECT fname, phone, fax, email
FROM members
WHERE lname = '$webform_lname';";
```
b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:
```
'; if (select user) = 'sa' waitfor delay '0:0:5'
```
c) The query that the web application will create and execute on the database becomes:
```
SELECT fname, phone, fax, email FROM members
WHERE lname = '';
if (select user) = 'sa' waitfor delay '0:0:5'
```

d) If the query takes more than 5 seconds to return results, the attacker can infer that the application is logged into SQL Server with system administrator privileges. This means the attacker can create and delete users, tables, and has privileges to execute any command against the database.

e) Using this technique, the attacker can ask yes/no questions [if yes, wait 5 seconds] to discover attributes of the backend database such as the version and patch level, or even if an injected command such as inserting a row into the users table executed successfully.

This type of attack demonstrates that configuring the production database server so that it does not display errors is insufficient to thwart a persistent attacker.

3.7 Alternate Encodings

A common defensive coding practice is to scan for characters such as SQL keywords, single quotation marks and SQL comment characters. Attackers can evade this security measure by using alternate methods to encode their attack strings, such as hexadecimal, ASCII, or Unicode characters [17]. Alternate encoding is used to obfuscate attacks that are launched using any of the other SQL injection techniques described in this section. For this reason, web applications that accept and process user data containing syntactic input are less secure than web applications that parameterize the user input so that it is treated as a literal at runtime – even if the framework does a thorough job sanitizing the input against all current known attacks and the framework is updated regularly to the latest version.

Another example of alternate encoding is described in an attack that is discussed on the website Internet Storm Center. This attack uses the `cast` command to update every `varchar` column in your database to append iframe text, which is an html tag that will display or download another document into an inline frame [37].

3.8 Second-order injection attacks

In second-order SQL injection attacks, a malicious user crafts the input to store values in the database to indirectly trigger an attack when the values are retrieved from the database and used to construct a query at a later time [17] [23] [36] [39] [49] [57].

Here is an example of a second-order SQL injection attack:

a) A web application allows users to register accounts with a username and password, using input entered into a web form. The application also allows users to change their passwords via a webform by entering their username, old password, and new password. The web application code builds the SQL command:

```
$query = "UPDATE USERS SET PASSWORD = '" + newpassword + "'
WHERE USERNAME = '" + username + "'
AND PASSWORD = '" + oldpassword + "'";
```

b) Earlier, a malicious user had created an account with a user name:

```
admin'--
```

c) Now, the malicious user visits the change password page and enters the username, along with the password previously entered and a new password: secret

d) The query that the web application will create and execute on the database becomes:

```
UPDATE USERS SET PASSWORD = 'secret'
WHERE USERNAME = 'admin'--'
```

e) The rest of the query is ignored because of the comment character "- - ". The malicious user has just changed the administrator password to "secret" and has full control of the database.

4. Using a Web Application Framework to Mitigate Development Risks

A web application framework provides a single method of accessing the database, rather than having database connections intermixed throughout the codebase. Some frameworks are so easy to use that they may enable programmers with limited understanding of database connections to build sophisticated applications quickly. They may give developers a false sense of security with regards to SQL injection vulnerabilities.

One important aspect to consider is whether the queries are constructed as parameterized queries. In a parameterized query, inputs (parameters) are supplied when the query is constructed. Parameters are treated as a literal at runtime and thus will not include syntactic content from the user input.

A framework that attempts to strip out individual characters but does not parameterize the query may leave the application vulnerable to an injection attack disguised with alternate encoding. In order to protect against all the types of SQL injection attacks, the application framework must ensure that no syntactic input is accepted from users.

Another aspect to consider is whether it is possible for programmers to connect to the database directly rather than through the framework. Many frameworks allow this capability for programmers to create custom SQL commands when very complex class models do not map easily to the framework's functionality. This flexibility may result in SQL injection vulnerabilities, and should be considered carefully before being implemented. Such custom code requires thorough testing.

4.1 Characteristic functionalities of a web application framework

Web application frameworks help protect developers from security vulnerabilities and centralize functions to avoid repetitive and non-standard code. They enable developers to produce new web applications faster, and handle an increasing volume of data, templates, and new business functionality requests.

A typical web application framework offers high-level abstractions of common Web development patterns such as Model-View-Controller, as well as streamlined workflows for frequent programming tasks and well-established conventions for solving common problems. This section describes some of the most common functionalities offered by web application frameworks.

4.1.1 Security

Most of the web application frameworks discussed in this section offer authentication and authorization modules either as part of the core package or optional modules. These services enable the application to identify the users and authenticate with passwords or other methods, and restrict access to functions, data, and other assets based on group membership or other defined criteria (authorization).

It is important to point out that authentication services offered by a web application framework manage users of the application, not users of the database. Typically the application logs into the database with an application login/password, and has a fairly high level of privilege. An application needs to be able to write to the database and needs read privileges on any information that it needs to present to any application user.

4.1.2 Database access and mapping

Most of the web application frameworks discussed in this section offer either a built-in API to access the database and filter user input in some way, or are compatible with an Object-Relational-Mapping (ORM) framework such as Hibernate.

There are several advantages to using a framework with built-in data access services or a persistence framework. The most obvious is that developers do not need to manually create data access classes, resulting in less custom code to test and maintain. This is because the framework provides the data access logic, basic data storage functionality and business object population. Another advantage is that application programmers do not need to know the details of the relational database. Because of the overhead involved with opening and closing database connections, the ability to manage pooled connections increases performance significantly.

The main responsibility of a data access layer is to insulate the business logic layer from the technical specifics of how the information is stored and accessed. Isolating the business logic layer from the technical specifics of how information is stored allows developers to achieve database access transparency.

Functions of the data access layer typically include following;

a) Map data between the business application and the physical data structure

b) Manage lookup methods and Create, Read, Update and Delete operations with the data store

c) Handle data-type conversions

d) Handle data access/retrieval related errors.

4.1.3 URL mapping

Most of the web application frameworks discussed in this section have a URL mapping functionality. This allows the framework to interpret URLs and direct the user to the correct page as it is generated by the page templating system and stored content. Rather than directing the user to a page address that resembles "/page.cgi?cat=science&topic=physics" the address that is displayed would be "/page/science/physics."

In addition to simplifying the site for users, it also allows better indexing by search engines. If the site is redesigned and implemented with another technology, all the links to the pages will still work. This is important as a site becomes more popular and has been linked to by other websites, and the address is printed in articles or on marketing collateral.

There are different ways to implement this functionality in a framework. For example, Django matches the provided URL against patterns using regular expressions. Other frameworks use a URL Rewriting service to translate between the URL displayed for users and the URL used internally within the framework. Other frameworks implement the URL mapping using a graph traversal technique.

4.1.4 Page template system

A page template system is a core component of a web application framework. A web page template is an HTML page with embed tags inserted where the content will appear. The purpose is to separate the presentation from the content. For a web-based catalog, rather than a separate static web page for each one of 500 products, there might be a single web page template connected through the web application framework to a database containing the content for each of the 500 products.

While it is often possible for the embed tags to support limited logic processing, like IF and FOREACH, standard practice is to use this kind of logic only for decisions that need to be made for the presentation

layer. In doing so, business application code is kept separated, in keeping with the Model-View-Controller design pattern.

4.1.5 Caching

Because caching is so important to improving performance, most web application frameworks provide some type of caching service. Caching web documents improves the response time of the system from a user perspective, and also reduces bandwidth and server load. A web cache service stores copies of documents passing through it, so that it can re-use the already-generated page for subsequent requests when appropriate.

4.1.6 The Model-View-Control design pattern

Most of the web application frameworks discussed in this section use the model-view-control (MVC) design pattern, a framework developed in the late 1970s by Trygve M. H. Reenskaug for the Smalltalk platform [14].

In this design pattern there are three roles:

a) Model – object containing all data and behavior that is not used for the User Interface

b) View – the display of the Model in the User Interface

c) Controller – takes user input, manipulates the Model, and updates the View

The MVC pattern encourages a decoupling of the model from the view, using the controller as an intermediate. This pattern produces a maintainable architecture because the user interface and the data handling can be modified independently from each other. The architecture is flexible because multiple user interfaces can share the same business model. Since the Model, View, and Controller can be implemented in different physical layers, the architecture is also scalable.

4.2 Common web application frameworks

There are hundreds of web application frameworks available. This thesis focuses on the most pertinent frameworks, that is, the frameworks that are popular now or likely to grow in popularity in the future. To determine which of the many options are important to discuss, the name of each framework was used as a search term in three different contexts described below. The complete results are reported Appendix 1. Table 1 contains information about the top scoring framework in each of 5 categories, with frameworks grouped according to the programming language they support.

Dice.com is an IT job board [2] that allows a user to search for job postings that request experience in particular frameworks. This is a snapshot measure of which frameworks are mentioned in the job ads, and thus which frameworks are currently being used in the industry. This is also a measure of how relevant a framework is in the job market.

Amazon.com is a large online bookseller, that allows a user to search for book titles within the “computers & internet” category [1]. A book contract is indicative that a framework has enough prospective users that a publisher expected to generate sufficient book sales to make a profit. More established frameworks represent a better risk/reward ratio for a publisher. It should be noted that the results of this search represent a lagging indicator, because of the length of time it takes to write and publish a book on a framework.

Search Y Combinator [7] is an independent project to build a search utility for Y Combinator's Hacker News. Hacker News is a discussion board sponsored by YCombinator, a venture capital firm. Technical startup founders discuss which technologies they are currently using or considering using to build new web-based companies. The archives were searched to count how many times a specific framework comes up in the discussion. This is a leading indicator, since these are new companies with no existing infrastructure to support, and this highly technical group may be considered a community of early adopters.

Once the three contexts were determined, the frameworks were grouped according to five development languages: ASP.NET, Java, PHP, Python and Ruby The name of each framework in the five groups was used as a search term in the three different search engines: Amazon.com, Dice.com, and SearchYC. The number of hits for each search term was recorded, and the percentage share of the total hits for the group was calculated. For example, Drupal received 1472 hits on SearchYC. This represented 40.89% of the total hits for all PHP frameworks discussed on Hacker News.

A weighted average was then calculated. By weighting the leading indicator slightly more heavily than the other two measures that indicate how established a framework has already become, the ranking that is produced is forward-leaning and tilted toward frameworks that are expected to increase in popularity in the near future. The percentage share for Search YC was weighted at 40%, and each of the percentage shares for Dice and Amazon were weighted at 30%. The frameworks were then sorted in descending order by the weighted average, to produce a ranked list of the most pertinent frameworks. Table 1 lists the framework with the highest score in each of the five groups. A full list of web application frameworks and their scores is available in Appendix 1.

Table 1— Percentage Share of Search Hits Returned for Pertinent Frameworks by Search Context

Language	Project	Weighted Avg.	Search YC		Dice		Amazon	
ASP.NET	ASP.NET MVC	87.83%	504	92.65%	633	88.78%	107	80.45%
Java	Spring	50.68%	494	23.83%	3099	86.98%	145	50.17%
PHP	Drupal	41.50%	1472	40.89%	230	34.64%	148	49.17%
Python	Django	70.65%	7324	84.36%	135	77.14%	61	45.86%
Ruby	Ruby on Rails	94.08%	15514	97.34%	628	96.02%	151	87.79%

4.3 Defining Object Relational mapping (ORM)

Object-relational mapping services convert data between incompatible type systems in relational databases and object-oriented programming languages and provide an object-based view of data to allow the application to specify Create, Read, Update, Delete operations without embedding SQL queries within the application. Mehta [30] provides the following list of basic criteria for good quality ORM tools:

a) *Object-to-database mapping:* An ORM tool used to map business objects to back-end database tables using metadata.

b) *Object caching:* Used to improve performance in the persistence layer

c) *Multiple database-platform support:* the layer of abstraction between the application and the database reduces the configuration rework required to accommodate changes to the physical implementation of the database.

d) *Dynamic querying:* the building of dynamic SQL queries based on user input

e) *Lazy loading:* the optimization of memory utilization for the database by prioritizing which components need to be loaded into memory when the program starts, and which need only be loaded if they are specifically requested. This improves performances in situations where the dependent components are never actually used.

f) *Nonintrusive persistence:* the business application calls the persistence API and passes the persistence objects to it, but the API code does not intrude into the persistence objects, which do not need to extend or inherit from any API function, class, or interface.

g) Stored procedure support: at times stored procedures are really the only viable option because of performance problems with long-running or complex queries in the ORM tool.

5. Analysis and Comparison of Frameworks' Data Access Methods

The data access method for each framework was analyzed, considering whether the framework creates parameterized queries or merely filters the user input, and whether there is a way to bypass the data access methods to create unsafe queries. These options are shown in the analysis flow found in Figure 3.

One important aspect considered is whether the queries are constructed as parameterized queries. In a parameterized query, inputs (parameters) are supplied when the query is constructed. Parameters are treated as a literal at runtime and thus will not include syntactic content from user input.

A framework that attempts to strip out individual characters but does not parameterize the query may leave the application vulnerable to an injection attack disguised with alternate encoding. In order to protect against all the types of SQL injection attacks, the application framework must ensure that no syntactic input is accepted from users.

Another aspect considered is whether it is possible for programmers to connect to the database directly rather than through the framework. Many frameworks allow this capability for programmers to create custom SQL commands. This flexibility may result in SQL injection vulnerabilities, and should be considered carefully before being implemented. Such custom code requires thorough testing.

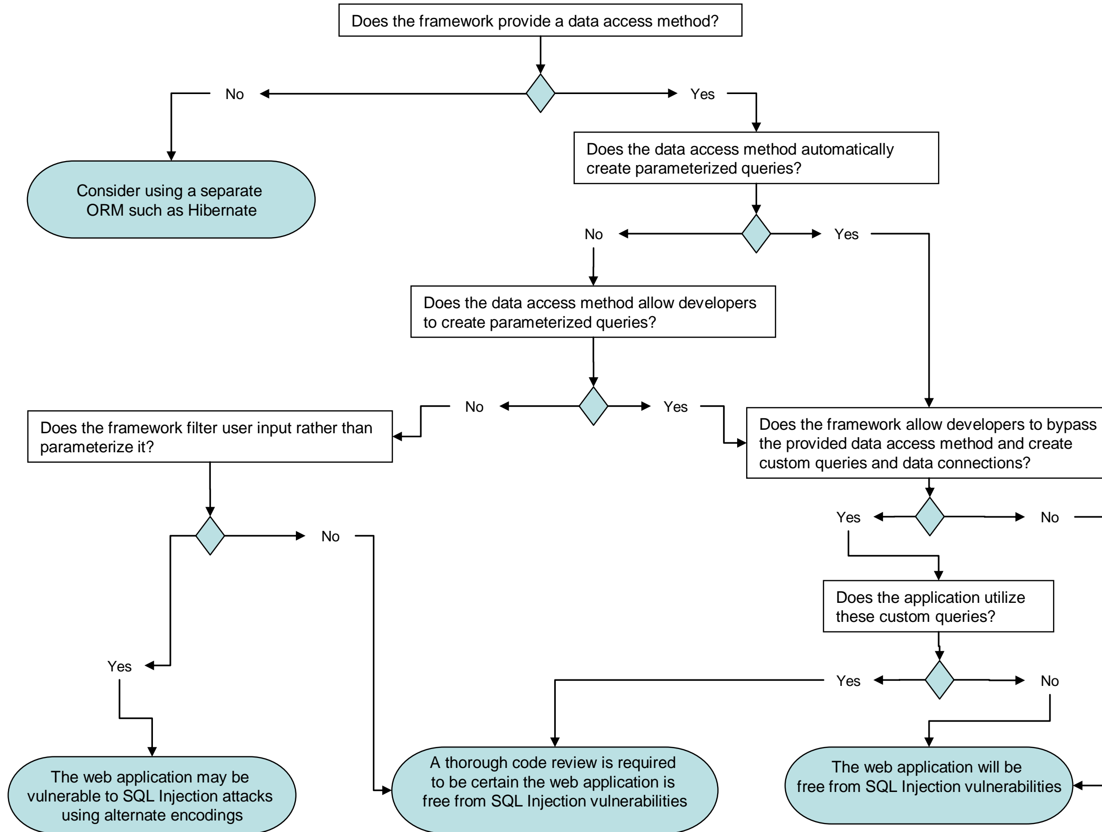

Figure 3— Analysis Flow of Web Application Frameworks

5.1 LINQ as part of the .NET Framework

Language-Integrated Query (LINQ) is a set of features introduced in Visual Studio 2008 and is also supported in Visual Studio 2010. LINQ provides standard patterns for querying and updating data in SQL Server databases, ADO.NET Datasets, and XML documents. It was released in 2007, is compatible with .NET version 3.5, the default version of .NET installed with the Windows 7 and Windows Server 2008 R2 operating systems. It is also supported in .NET version 4.

LINQ to SQL is an API (Application Programming Interface) which allows the programmer to access and manipulate data stored in a SQL Server database. As shown in Figure 5, LINQ to SQL is the point of access from the Business Tier to the Relational Database in the Data Tier. To work with other databases such as Oracle, third party .NET data providers will need to be used.

Prior to the introduction of LINQ, queries were expressed as simple strings which could not be checked at compile time. If the SQL query strings were formed by concatenating user input, an SQL injection vulnerability was introduced to the application. Now, the LINQ provider for SQL Server allows the .NET framework to automatically generate the SQL commands that will run on the database server, preventing the introduction of SQL injection vulnerabilities.

The ASP.NET MVC Framework is a web application framework based on ASP.NET and released in 2009 under the Microsoft Public License. The framework implements the model-view-controller pattern.

5.1.1 LINQ architecture

The technology stack diagram (Figure 4) shows the compatible technologies required for an example web application using .NET and LINQ. Connections to the SQL Server database are expected to go through LINQ but it is also possible to connect directly to the database using one of the .NET programming languages C# or VB.NET. In contrast to other example architectures in this thesis, the ASP.NET MVC / LINQ architecture is built on proprietary technology stack.

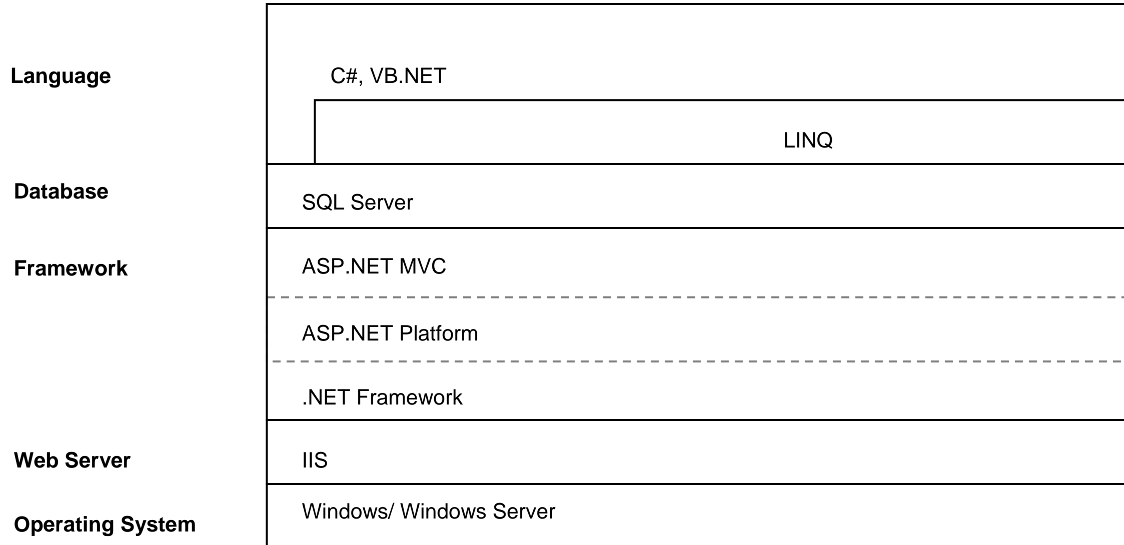

Figure 4— ASP.NET MVC with LINQ Technology Stack

The architecture diagram in Figure 5 shows a logical view of the technology architecture of an example web application using ASP.NET MVC and LINQ. The color coding shows the functionalities that are provided by the .NET framework and ASP.NET MVC in yellow, and the non-standard, highly customized parts of the application that must be built by the developers in purple. The pre-built functionality provided by the web application framework makes it easier and faster to build a fully functioning database-backed web application.

The expected method of accessing the database is through the data access layer provided by LINQ. However, the purple line in Figure 5 indicates that it is possible for developers to create custom code which accesses the database directly.

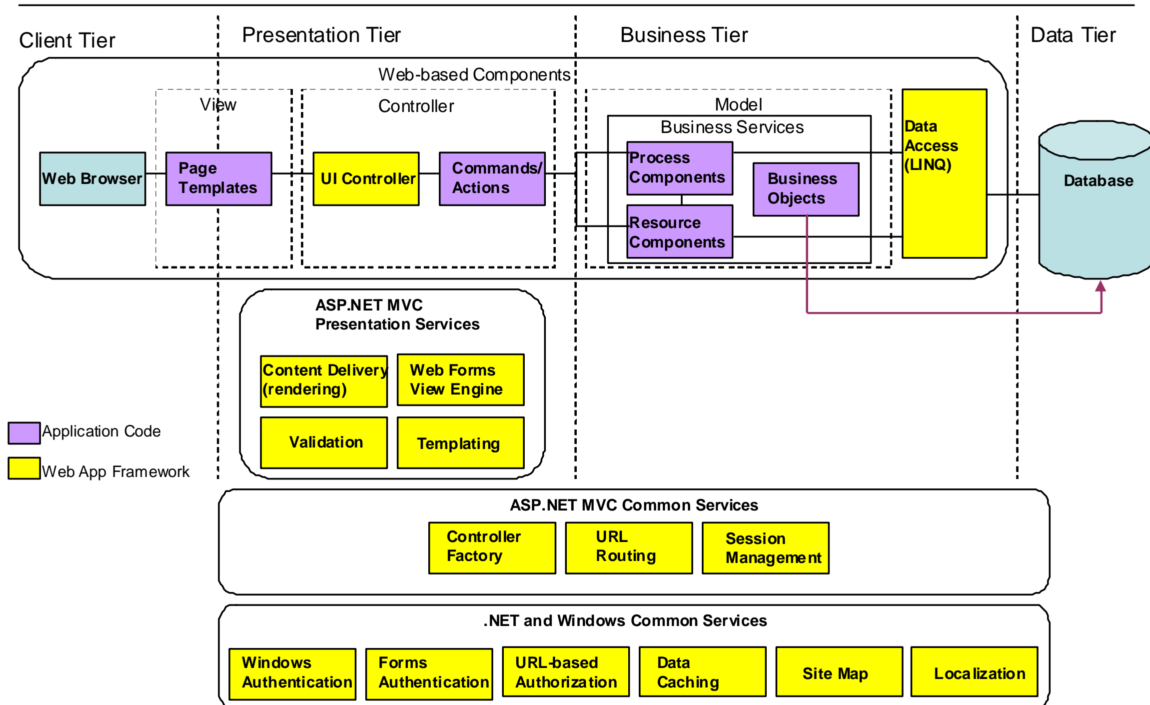

Figure 5— ASP.NET MVC with LINQ Logical Technology Architecture Diagram

5.1.2 How does LINQ query the database?

At runtime, queries specified in LINQ syntax in the application code are translated into SQL, and executed on the database [19] [22] [24] [28] [30] [33] [38] [43]. The automatically generated SQL query is recorded in the application log if the logging configuration is set up to do so. The query results are returned to the application as objects.

Here is an example of an SQL injection attack attempt.

a) The web application code specifies the query with the following LINQ syntax:

```
GridView1.DataSource =
from member in db.members
where member.LastName ==
txtLname.Text
select member;
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS—
```

c) The automatically generated query that the web application LINQ will create and execute on the database is:

```
SELECT [t0].[FirstName], [t0].[Phone],
[t0].[Fax], [t0].[Email]
FROM [dbo].[Members] AS [t0]
WHERE [t0].[LastName] = @p0
```

d) The automatically generated query is a parameterized query. Input by the user is represented by "@p0" above, and treated as a literal value when executed as part of the SQL statement. In other words, a literal value is considered to be semantic rather than syntactic content. Any commands included will not be executed. In the example above the value of @p0 is a string: `smith' UNION SELECT visacardno FROM PAYMENTS—`

e) The query will return no rows.

Secure software development expert Jason Schmitt argues [46] that exclusive use of LINQ to SQL for data access allows developers to create applications that are impervious to SQL injection, because of the following features:

- Automated use of parameterized queries: A parameterized query differs from a simple query in that parameters are supplied when the query is constructed. Parameters are treated as a literal at runtime. This obviates the need to sanitize user input. The SQL command that is generated will not include syntactic content from the user input. Because the process is automated, the use of parameterized queries is enforced for every point at which the application accesses the database using LINQ to SQL.

- Compile-time syntax checking: The great disadvantage of building SQL commands with quoted strings is that the commands are not recognized by the compiler, so syntax problems cannot be detected until runtime. LINQ to SQL objects are recognized and checked by the compiler.

5.1.3 Analysis

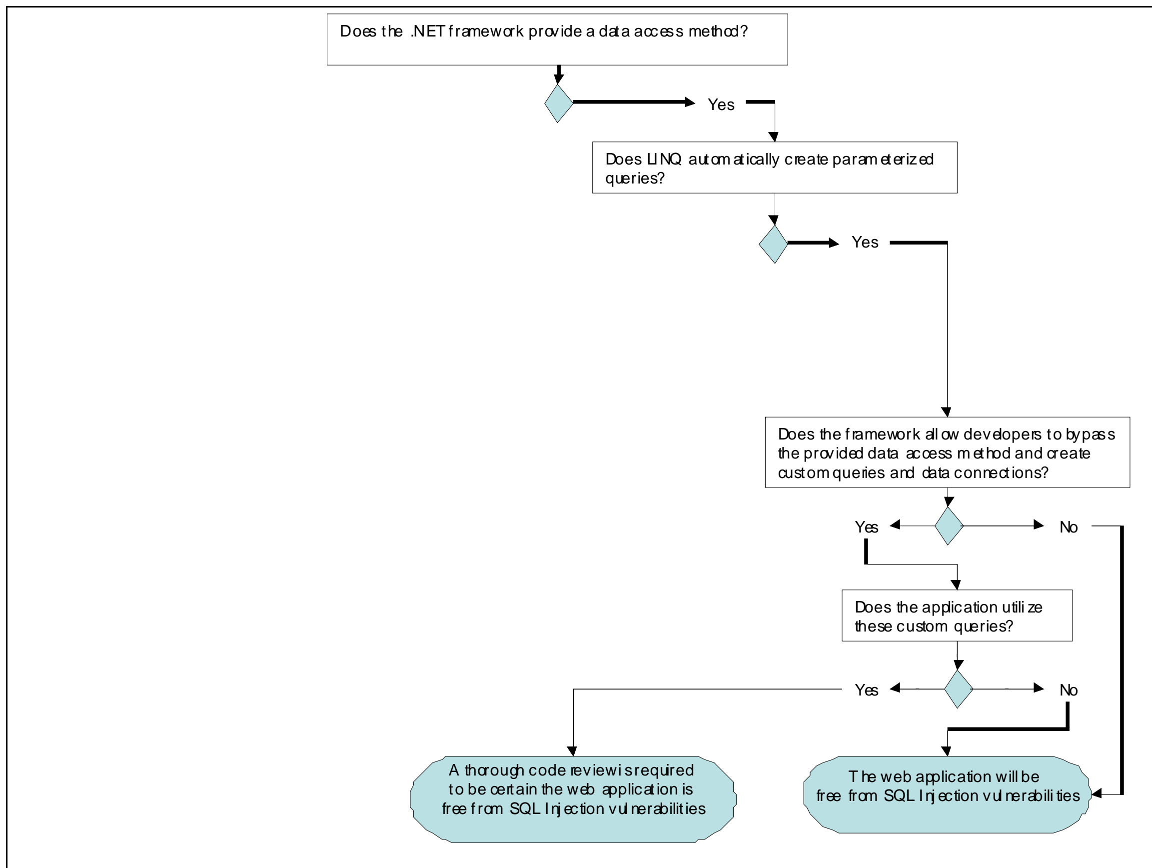

Figure 6— ASP.NET MVC with LINQ Data Access Analysis

Compared to the data access methods provided by other web application frameworks, LINQ to SQL does a better job of making it easier and more likely that a developer will create an application without SQL injection vulnerabilities. As shown in Figure 6, LINQ automatically creates parameterized queries, which leads to an application free from SQL injection vulnerabilities so long as custom queries are not used. LINQ to SQL translates the queries into parameterized SQL queries (in text form) and sends them to the SQL server for processing [33]. Visual Studio also provides a user interface for implementing many of the features of LINQ to SQL, called the Object Relational Designer.

LINQ queries can be used alongside non-LINQ queries in existing projects, so it is important to standardize coding practices so that unsafe data access methods are not introduced. Also, as shown in Figure 6, there is a method of creating custom queries. This refers to the ExecuteQuery method [33] which allows programmers to execute a SQL query, and then convert the result directly into objects. In these cases it is possible to create unsafe queries, so it is important to use parameterized queries. Parameters are expressed in the query text by using the same curly notation used by String.Format(). Then, String.Format() is called on the query string, substituting the curly braced parameters with generated parameter names such as @p0.

5.2 Active Record design pattern

Active Record is an Object Oriented design pattern [14] in which an object carries both data and behavior with persistent data stored in a database. The Active Record design pattern puts data access logic in the domain object (does not use a data access layer). However, the implementations of ActiveRecord in Ruby on Rails and CodeIgniter do resemble a data access layer.

The Active Record design pattern approach is as follows:

a) Create a domain model in which classes match the record structure of the database tables:
   1) Each field in a class matches a column in a table
   2) Each field types matches a SQL interface
   3) Each active record is responsible for:
      i) Saving and loading to the database
      ii) Performing domain logic on the data

b) Typical active record class methods include:
   1) Constructor to create a new instance of active record from the SQL result set row
   2) Constructer to create a new instance of active record to insert a row into a table
   3) Finder methods to wrap queries and return active record objects
   4) Methods to persist data from the object to the table
   5) Get and set methods to get and set fields in object
   6) Methods to perform business logic

The Active Record design pattern works best with CRUD (Create, Read, Update, Delete) database operations when the domain logic is fairly straightforward.

5.2.1 ActiveRecord as part of Ruby on Rails

Ruby on Rails is an open-source web application framework created in 2003 by David Heinemeier Hansson and extended by the Rails core team. Development for Ruby on Rails websites is done using the Ruby programming language. Ruby on Rails is based on the Model-View-Controller (MVC) architectural pattern, and provides tools such as scaffolding to automatically create some of the models and views needed for a website.

ActiveRecord is a core functionality of Ruby on Rails. It is an object-relational mapping (ORM) library that connects to databases, maps tables, and manipulates data. Using ActiveRecord, the developer can specify CRUD (create, read, update, delete) operations without explicitly using SQL.

5.2.2 Ruby on Rails Architecture

The technology stack diagram (Figure 7) shows the range of compatible technologies required for an example web application using Ruby on Rails. Connections to the database are expected to go through ActiveRecord but it is also possible to connect directly to the database using the Ruby programming language. As opposed to the ASP.NET/LINQ example, Ruby on Rails does not require a particular proprietary technology stack. Instead it supports proprietary operating systems like Windows and Mac

OSX and proprietary database management systems like SQL Server and Oracle, as well as open source technology stacks like Linux/Apache/MySql.

Language	Ruby
	ActiveRecord
Database	MySQL, PostgreSQL, SQLite, Oracle, SQL Server, DB2
Framework	Ruby on Rails
Web Server	Apache, lighttpd, or nginx proxying to Mongrel (or using FastCGI)
Operating System	Windows Server/ Linux/ Unix/ Mac OSX…

Figure 7— Ruby on Rails Technology Stack

The architecture diagram in Figure 8 shows a logical view of the technology architecture of an example web application using Ruby on Rails. The color coding shows the functionalities that are provided by Ruby on Rails in yellow, and the non-standard, highly customized parts of the application that must be built by the developers in purple.

The expected method of accessing the database is through the data access layer provided by the Rails implementation of ActiveRecord. However, the purple line in Figure 8 indicates that it is possible for developers to create custom code which accesses the database directly.

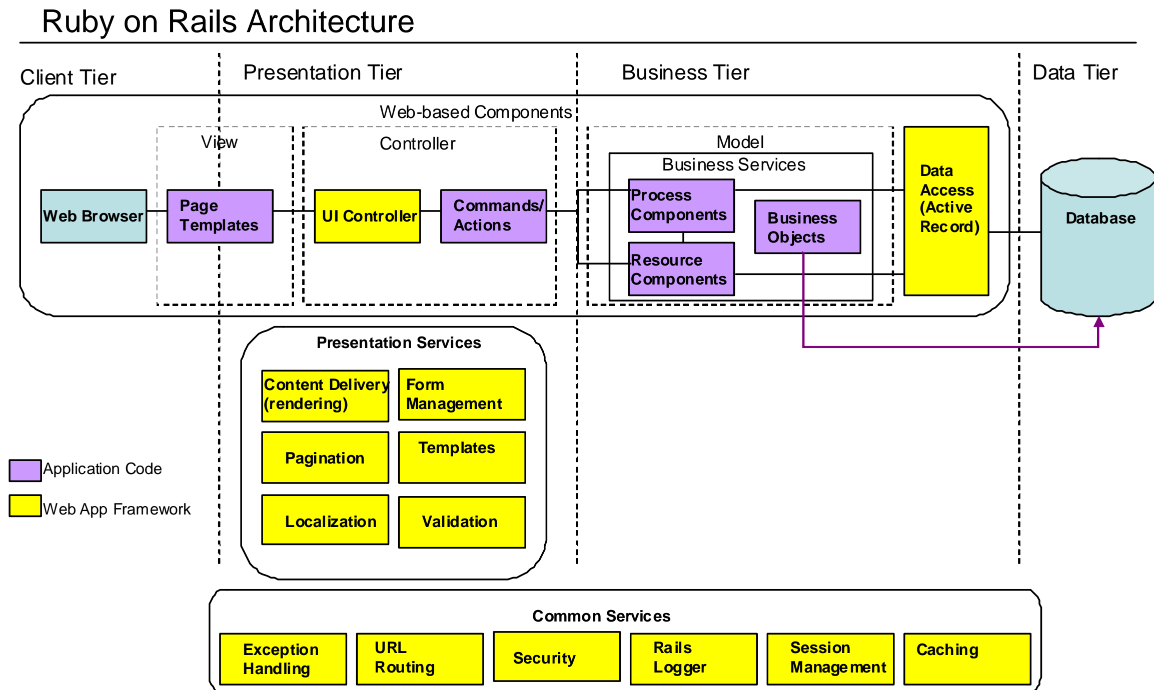

Figure 8— Ruby on Rails Logical Technology Architecture Diagram

5.2.3 How Does Rails ActiveRecord Query the Database?

It is possible to construct unsafe code within Ruby on Rails if the developer uses the conventional Ruby #{…} mechanism to substitute user input into a statement that contains conditions, limits, or SQL [16] [42] [51].

Here is an example of a SQL injection attack attempt.

a) The web application code specifies the query with the following ActiveRecord syntax, placing the arguments directly inside the condition string:

```
Members.find(:all, "lname  = '#{params[:lname]}'")
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS—
```

c) The resulting statement would resemble:

```
Members.find(:all, "lname = 'smith' UNION SELECT visacardno
FROM PAYMENTS --'")
```

d) The new query will display the information stored in the 'visacardno' column for all the rows in the "payments" table. When the results are not restricted by a "where" clause, all results are returned.

```
1234123412341234
3456345634563456
4567456745674567
```

Here is an example of an unsuccessful SQL injection attack attempt.

a) The web application code specifies the query with the following ActiveRecord syntax:

```
Members.find(:all, [lname = "?", params[:lname]])
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS—
```

c) The embedded placeholder "?" will be replaced at runtime with the correct value.

d) The query will return no rows.

5.2.4 Analysis

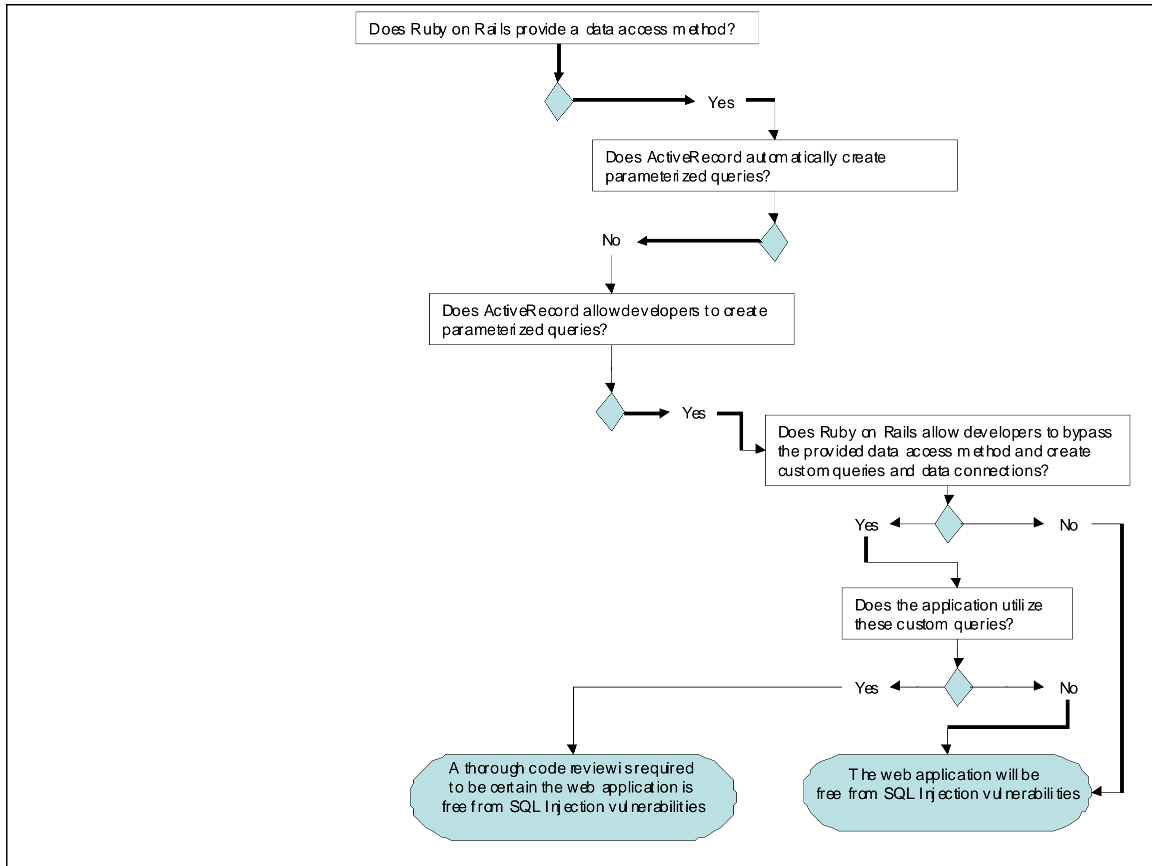

Figure 9— Ruby on Rails Data Access Method Analysis

Compared to the data access methods provided by other web application frameworks, the ActiveRecord syntax provided by Ruby on Rails allows the creation of unsafe queries as easily as safe queries. The danger presented by poor data access code is well documented. The warning in the documentation [16] is quite plain: "If you do this, you put your entire database at risk because once a user finds out he or she can exploit your database they can do just about anything to it. Never ever put your arguments directly inside the conditions string." Despite this warning, this is a huge risk for developers who have not read the documentation.

The risk is greater than the risks posed by the custom query method shown in Figure 9, because custom queries are typically used by more advanced developers creating complex applications. In Ruby on Rails, this refers to the `find_by_sql` method.[16] The `find_by_sql` method allows custom calls to the database and retrieves instantiated objects. It returns an array of objects even if the underlying query returns just a single record.

When used correctly according to the documentation, ActiveRecord as implemented by Ruby on Rails can help developers create web applications that are free from SQL injection vulnerabilities. The explicit warning in the documentation also helps raise awareness of the dangers of not using parameterized queries.

5.2.5 ActiveRecord as part of CodeIgniter

CodeIgniter is an open source web application framework released in 2006 by Ellis Lab, and is loosely based on the Model-View-Controller architectural pattern. Development for CodeIgniter websites is done using the PHP scripting language.

Although Drupal and Zend are much more popular PHP web application frameworks according to the findings in Appendix 1, CodeIgniter is included in this discussion because it is an interesting contrast to the implementation of ActiveRecord in Ruby on Rails. CodeIgniter uses a modified version of the Active Record Database Pattern. In contrast to Ruby on Rails, CodeIgniter attempts to mitigate the risk of SQL Injection vulnerabilities by automatically escaping the user input when creating dynamic queries. Query syntax is generated by the database adapter.

5.2.5.1 CodeIgniter Architecture

The technology stack diagram (Figure 10) shows the range of compatible technologies required for an example web application using CodeIgniter. Connections to the database are expected to go through ActiveRecord but it is also possible to connect directly to the database using PHP. As opposed to the ASP.NET/LINQ example, CodeIgniter does not require a particular proprietary technology stack. Instead it supports proprietary operating systems like Windows and Mac OSX and proprietary database management systems like Oracle, as well as open source technology stacks like Linux/Apache/MySql.

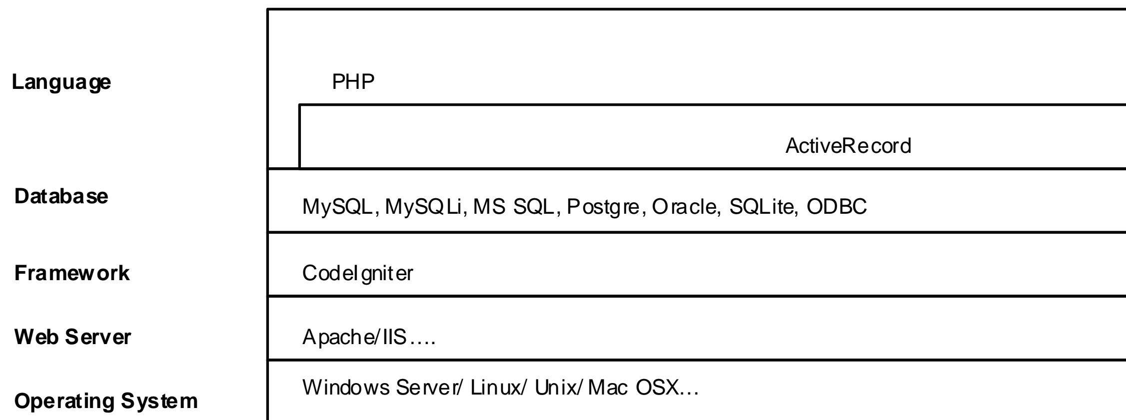

Figure 10 — CodeIgniter Technology Stack

The architecture diagram in Figure 11 shows a logical view of the technology architecture of an example web application using CodeIgniter and Active Record. The color coding shows the functionalities that are provided by CodeIgniter in yellow, and the non-standard, highly customized parts of the application that must be built by the developers in purple.

The expected method of accessing the database is through the data access layer provided by the CodeIgniter implementation of ActiveRecord. However, the purple line in Figure 11 indicates that it is possible for developers to create custom code which accesses the database directly.

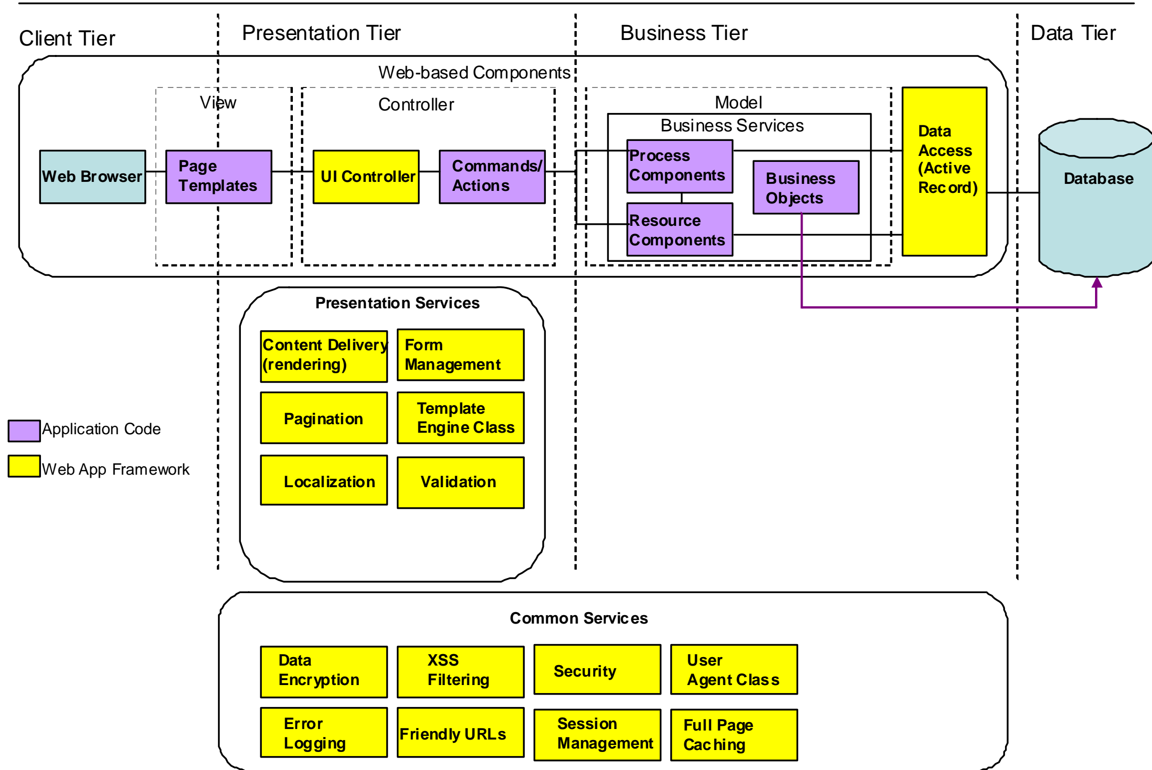

Figure 11 — CodeIgniter Logical Technology Architecture Diagram

5.2.5.2 How Does CodeIgniter ActiveRecord Query the Database?

The database library in CodeIgniter enables the application code to pass in simple SQL queries [53]. It also allows developers to bypass the built-in Active Record patterns and send a complex query directly to the database; this could result in an SQL injection vulnerability if security policies are not followed properly.

Here is an example of an unsuccessful SQL injection attack with code that follows the ActiveRecord syntax:

a) The web application code specifies the query with the following ActiveRecord syntax:

```
$this->db->select('fname, lname, phone');
$this->db->from('Members');
$$this->db->where('lname', $lname);
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS—
```

c) All values passed to this function are escaped automatically. The query syntax is generated by the database adapter.

d) The query will return no rows.

5.2.6 Analysis

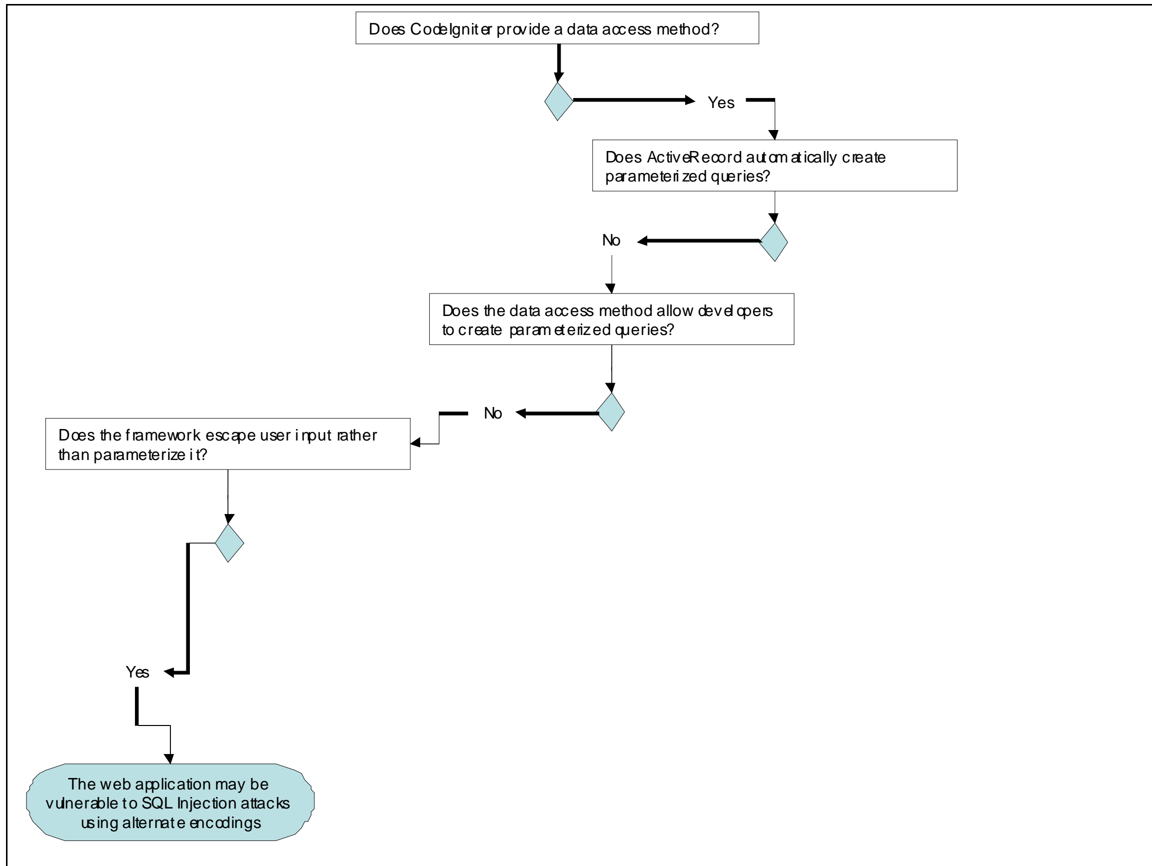

Figure 12 — CodeIgniter Data Access Method Analysis

Even though Ruby on Rails and CodeIgniter both use the Active Record design pattern, they are implemented differently and do not provide the same level of protection against SQL injection vulnerabilities. CodeIgniter does not provide a way to create parameterized queries using Active Record or its other built-in data access methods. While the framework does offer automated filtering of known malicious user entries, it may not protect against all possible encoding and obfuscation of attacks. This is shown in Figure 12.

CodeIgniter offers two built-in data access methods: ActiveRecord and Query Bindings. Both methods handle SQL injection the same way – by using their built-escaping classes to recognized datatype and add the appropriate escape characters for the database system (for strings it is usually `'` and `'`) [52].

In addition, CodeIgniter offers a method $.this->db->escape() to escape custom queries[12]. Developers are also free to create custom queries without using the escape method, and the documentation [12], in contrast to the Ruby on Rails documentation's strong reproach, the CodeIgniter documentation for `$this->db->escape()` contains only this advice for developers: "It's a very good security practice to escape your data before submitting it into your database."

When developers use the `escape()` method, Query Bindings or Active Record, CodeIgniter can help developers create web applications that are protected from known attack methods. Frequent framework updates will be required to keep a web application free from new SQL injection attack methods as they arise. Even with due vigilance, there will be time windows of vulnerability between the discovery of an innovative attack method and the updated framework response.

There is a risk that developers may create custom queries without using one of these methods, and the documentation is not explicit in warning of the danger of SQL injection vulnerabilities. Multiple methods of database access may also make a thorough code review more difficult. Creating and enforcing coding standards for database access can help.

5.3 Escape Filtering as part of Django

Django is an open source web application framework released in 2005, and maintained by Django Software Foundation since 2008. It was originally developed to manage several news-oriented sites for The World Company of Lawrence, Kansas. It is based on the model-view-controller architectural pattern. Development for Django websites is done using the Python programming language.

5.3.1.1.1 Django Architecture

The technology stack diagram (Figure 13) shows the range of compatible technologies required for an example web application using Django. Connections to the database are expected to go through Django's data access layer but it is also possible to connect directly to the database using custom code. Similarly to other open source web application frameworks, Django does not require a particular proprietary technology stack. Instead it supports proprietary operating systems like Windows and Mac OSX and proprietary database management systems like Oracle, as well as open source technology stacks like Linux/Apache/MySql.

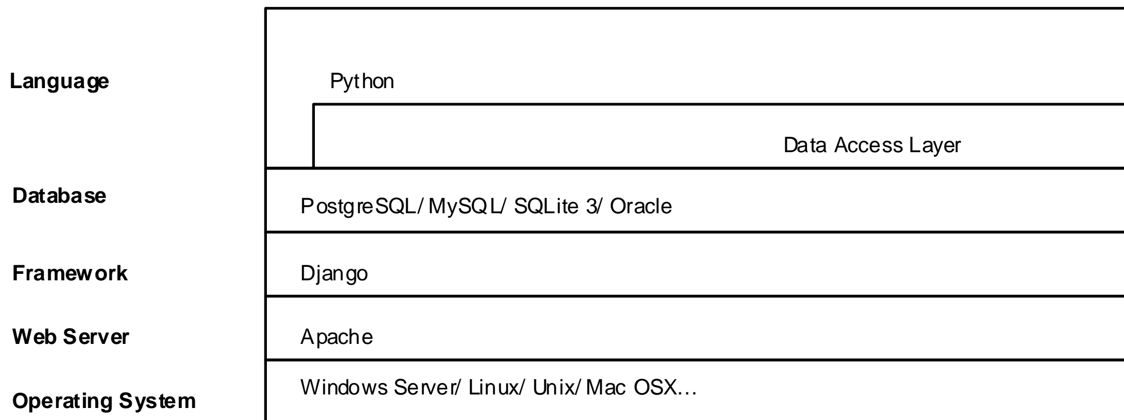

Figure 13 — Django Technology Stack

The architecture diagram in Figure 14 shows a logical view of the technology architecture of an example web application using Django. The color coding shows the functionalities that are provided by Django in yellow, and the non-standard, highly customized parts of the application that must be built by the developers in purple.

The expected method of accessing the database is through the data access layer provided by Django. However, the purple line in Figure 14 indicates that it is possible for developers to create custom code which accesses the database directly.

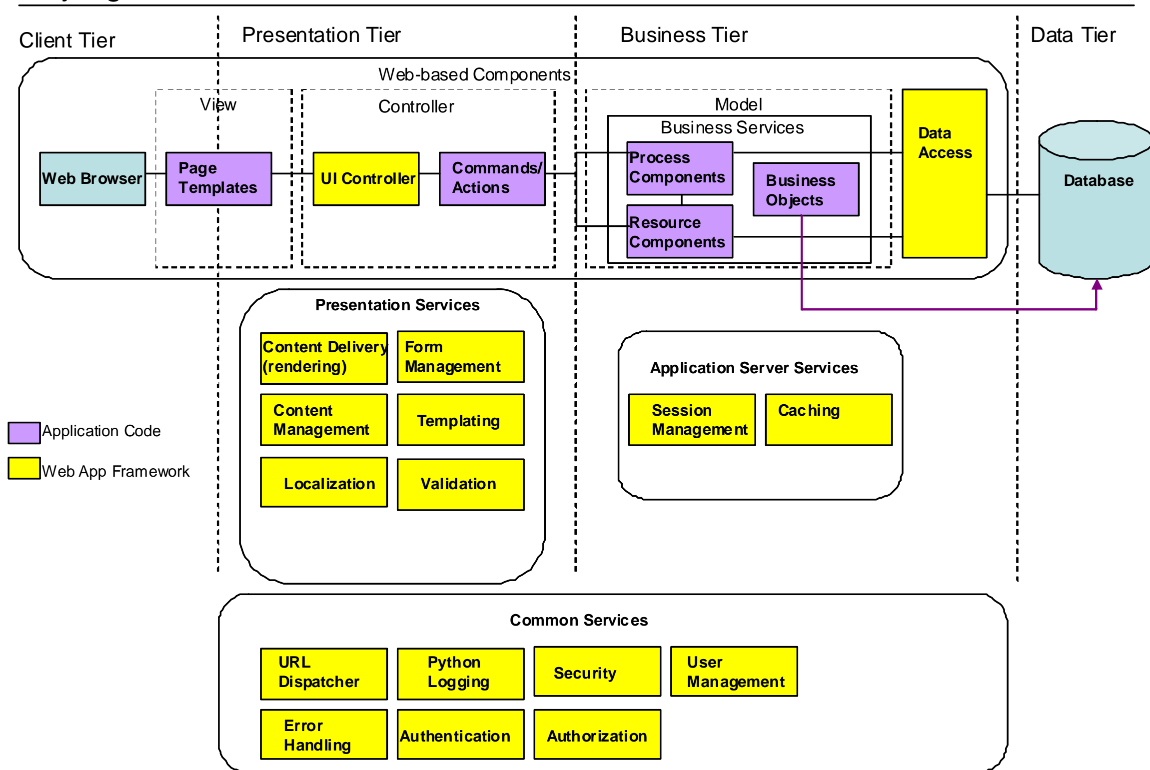

Figure 14 — Django Logical Technology Architecture Diagram

5.3.2 How Does Django Query the Database?

Django users are advised [18] to use Django's filtering ability to escape malicious input using the escape characters corresponding to the relational database management system defined for the application (MySQL, etc). This will only work if the encoding used in the malicious input is recognized by the filter in order to be properly escaped. The query is still built using user input, rather than parameterized. Because the framework accepts and executes syntactic data, the risk remains that the filter will be insufficient to thwart a particularly novel, obfuscated attack.

Here is how the filter is called:

In the view:

```
def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = ( Q(members__last_name__icontains=query)
        )
        results = members.objects.filter(qset).distinct()

    else:
        results = []
    return render_to_response("members/search.html", {
        "results": results,
        "query": query
    })
```

In the HTML form:

```
<form action="." method="GET">
    <label for="q">Search: </label>
    <input type="text" name="q" value="{{ query|escape }}">
    <input type="submit" value="Search">
  </form>
```

It is also possible in Django to create unsafe database calls by hand from user input:

```
def members(request):
    memberlist = request.GET[lname]
    sql = "SELECT * FROM members WHERE lname = '%s';" % username
```

.

Here is an example of a successful SQL injection attack attempt.

a) The web application code specifies the query:

```
def members(request):
    memberlist = request.GET[lname]
    sql = "SELECT * FROM members WHERE lname = '%s';" %
username
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS—
```

c) The resulting statement would resemble:

```
SELECT * FROM members WHERE username = ' smith' UNION
SELECT visacardno FROM PAYMENTS
```

d) The new query will display the information stored in the 'visacardno' column for all the rows in the "payments" table. When the results are not restricted by a "where" clause, all results are returned.

```
1234123412341234
3456345634563456
4567456745674567
```

The solution to this vulnerability is to construct the query with bind parameters.

Here is an example of an unsuccessful SQL injection attack attempt.

a) The web application code specifies the query using bind parameters instead of string interpretation:

```
def members(request):
    memberlist = request.GET['lname']
    sql = "SELECT * FROM members WHERE lname = %s;"
    cursor = connection.cursor()
    cursor.execute(sql, [memberlist])
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS—
```

c) The low-level `execute` method takes a SQL string with `%s` placeholders and inserts parameters from the list passed as the second argument.

d) The query will return no rows.

It is possible to construct the query with bind parameters, but bypassing the data access layer is not the best policy. Also, multiple methods to access the database may make a thorough code review more difficult, and coding standards more difficult to enforce.

5.3.3 Analysis

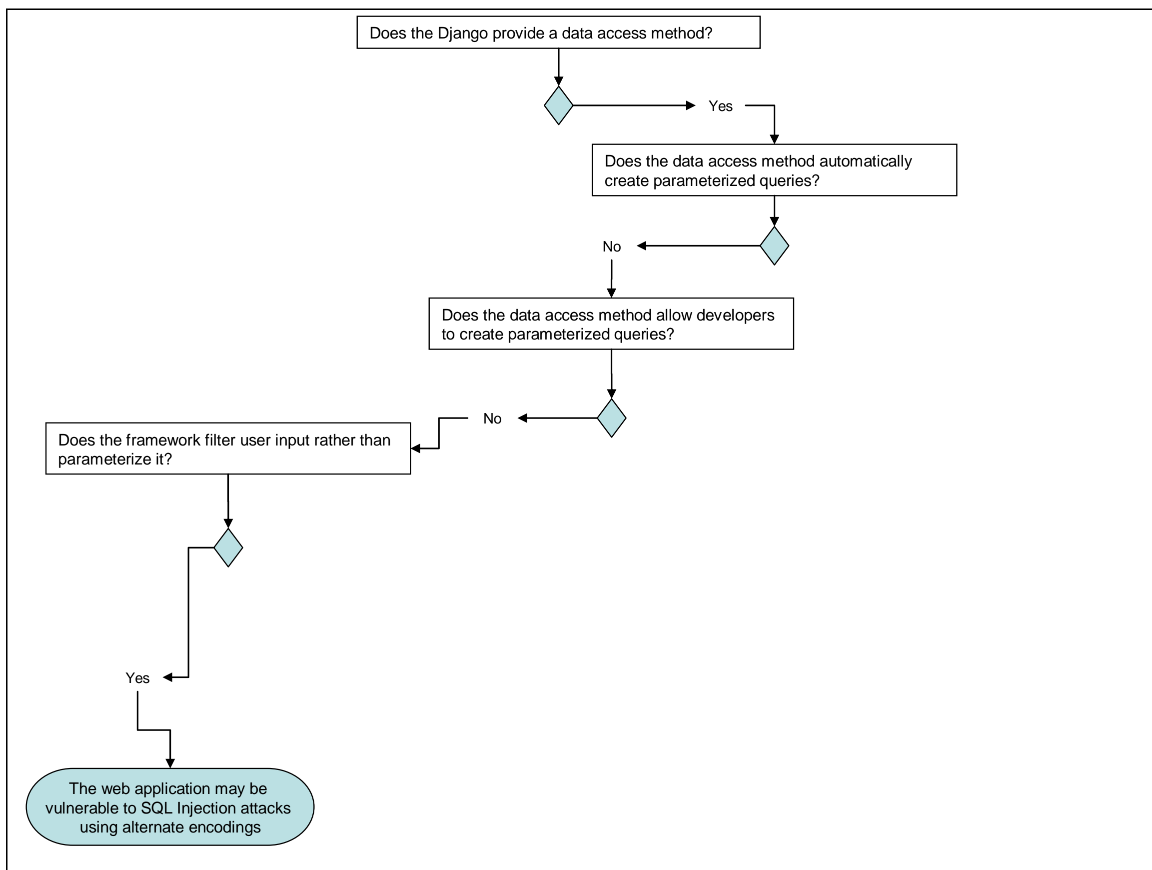

Figure 15 — Django Data Access Method Analysis

Compared to the data access methods provided by other web application frameworks, Django does not provide a built-in data access method that automatically creates parameterized queries. Instead, Django's database API automatically escapes all special SQL parameters, according to the quoting conventions of the database server used in the application (PostgreSQL/MySQL/SQLite 3/Oracle)[18]. This is shown in Figure 15.

Django also provides the `raw()` method to create custom queries. There is a potential to create unsafe queries using the `raw()` method, and this is clearly spelled out in the documentation[11]: "Warning! Do

not use string formatting on raw queries! If you use string interpolation, sooner or later you'll fall victim to SQL injection." Instead, developers are advised to always use the params list to create a parameterized query which is safe from SQL injection vulnerabilities.

Django's database API can help developers create web applications that are protected from known attack methods. Frequent framework updates will be required to keep a web application free from new SQL injection attack methods as they arise. Even with due vigilance, there will be time windows of vulnerability between the discovery of an innovative attack method and the updated framework response. While it is possible to create customized parameterized queries each time the application needs to accept user input, it is not considered a best practice to routinely bypass the data access layer. With multiple ways of accessing the database used intermittently in the code, a thorough code review becomes more difficult.

5.4 DB_Query as part of Drupal

Drupal is an open-source web application framework released in 2001 by Dries Buytaert. Development for Drupal websites is done using the PHP scripting language.

Drupal's database abstraction provides a level of abstraction so that the code is not specific to one database. Most Drupal database queries are performed by a call to `db_query()`, which executes a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement to the active database connection [55]. Drupal passes all queries as prepared statement strings, described in the documentation[50] as a "template" of a query that omits literal or variable values in favor of placeholders. The values to place into those placeholders are passed separately, and the database driver handles inserting the values into the query. The term "prepared statement" is sometimes used interchangeably with the term parameterized query[58].

5.4.1 Drupal Architecture

The technology stack diagram (Figure 16) shows the range of compatible technologies required for an example web application using Drupal. Connections to the database are expected to go through Drupal's data abstraction layer but it is also possible to connect directly to the database using custom code. Similarly to other open source web application frameworks, Django does not require a particular proprietary technology stack. Instead it supports proprietary operating systems like Windows and Mac OSX and proprietary database management systems like Oracle, as well as open source technology stacks like Linux/Apache/MySql.

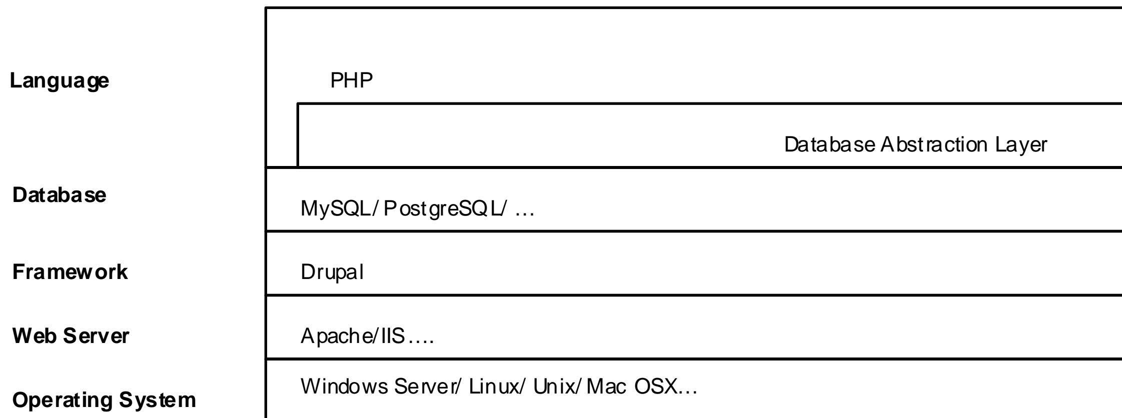

Figure 16 — Drupal Technology Stack

The architecture diagram in Figure 17 shows a logical view of the technology architecture of an example web application using Drupal. The color coding shows the functionalities that are provided by Drupal and PHP in yellow, and the non-standard, highly customized parts of the application that must be built by the developers in purple.

The expected method of accessing the database is through Drupal's `db_query()` method. However, the purple line in Figure 17 indicates that it is possible for developers to create custom code which accesses the database directly.

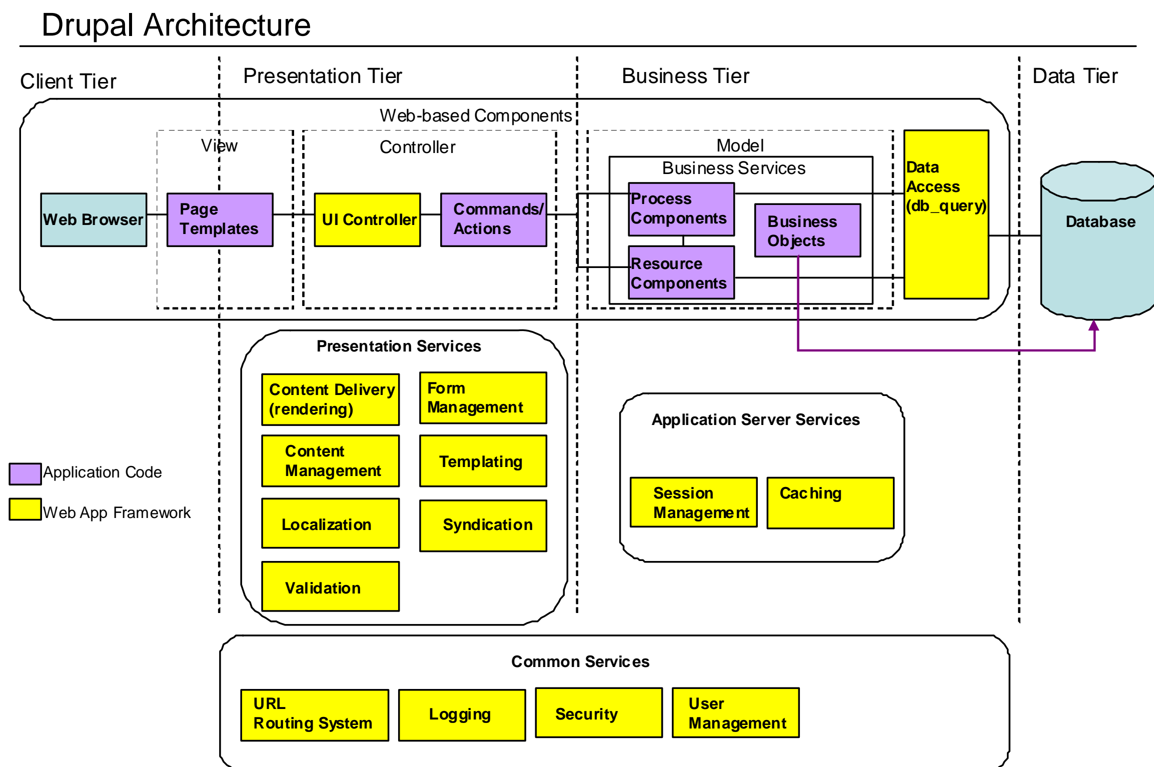

Figure 17 — Drupal Logical Technology Architecture Diagram

5.4.2 How Does Drupal Query the Database?

Most Drupal database queries are performed by a call to `db_query().` The function accepts up to three parameters: `$query` `$args` and `$options`. The first parameter, `$query,` accepts the prepared statement query to run, using named placeholders as shown in the example below. The second parameter, `$args,` accepts an associative array of values to substitute into the query as indicated by the placeholders. The last parameter (not shown in the example) is the `$options` parameter. This parameter is not required, and may be used to alter how the query operates.

Here is an example of an SQL injection attack attempt.

a) The web application code specifies the query performed by a call to `db_query()`:

```
db_query('SELECT * FROM {members} m WHERE m.lname =
:lname');
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS
```

c) ":lname" is a placeholder that will be replaced with a literal value when the query is executed. In this example the value of ":lname" is a string:

```
smith' UNION SELECT visacardno FROM PAYMENTS
```

d) The query will return no rows.

5.4.3 Analysis

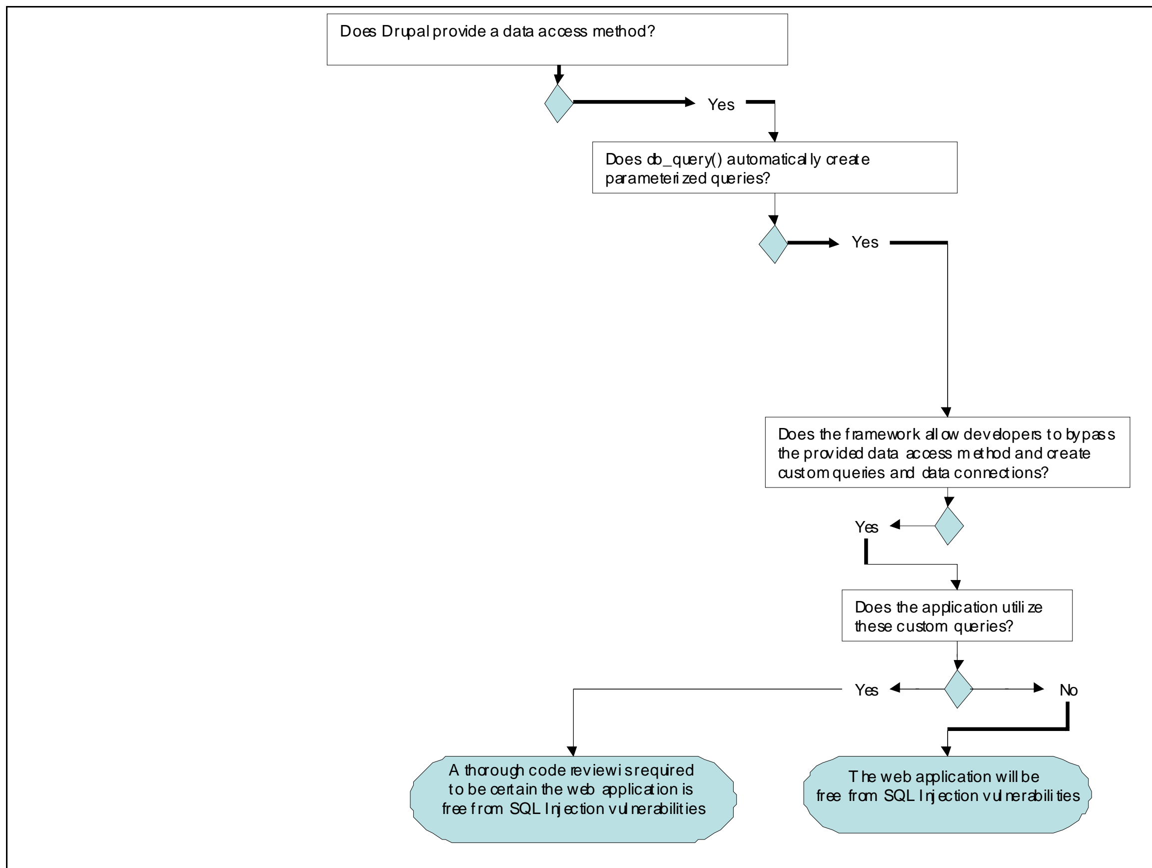

Figure 18 — Drupal Data Access Method Analysis

Compared to the data access methods provided by other web application frameworks, Drupal does a better job of making it easier and more likely that a developer will create an application without SQL injection vulnerabilities. In Figure 18 it shows that Drupal automatically creates parameterized queries, which leads to an application free from SQL injection vulnerabilities so long as there are no queries which bypass the database abstraction layer.

When developers call `dbquery()`, they pass to it a prepared statement query with placeholders to mark where a literal will be inserted into a query for execution. The values for the placeholders (which are not escaped or quoted regardless of their type) are passed separately to differentiate between SQL syntax and user-provided values[50]. In this way, the framework prevents syntactic data from user input from being accepted and executed, creating an application that is free from SQL injection vulnerabilities.

As noted in Figure 18, there is also a method of creating custom queries. Drupal's database abstraction layer is built atop PHP's PDO (PHP Data Objects) database API and inherits much of its syntax and semantics. Unfortunately (from a SQL injection attack point of view), this also allows developers to issue queries directly in PHP, which may result in unsafe queries.

5.5 Hibernate: An Object/relational mapping (ORM) framework

Hibernate is an object-relational mapping (ORM) library for the Java language started in 2001 by Gavin King, and currently supported by Red Hat and distributed under the GNU Lesser General Public License. Hibernate provides a framework mapping Java classes to database tables and can be used with web application frameworks such as Spring, the most popular of the frameworks for the Java programming language (see Appendix 1).

Spring and Hibernate are very commonly used together. An Amazon[1] search for both terms together turns up 85 books, compared to 145 books for Spring as a single search term. Many Spring books contain a section or chapter on integration with Hibernate.

5.5.1 Hibernate Architecture

As seen in Figure 19, Hibernate relies on the Java Database Connectivity (JDBC) API to connect to the database. The figure shows Hibernate being used with Plain Old Java Objects (POJOs), as opposed to entity beans or other more constrained Java conventions. Hibernate uses a small XML mapping file to determine which tables relate to which objects. The configuration file is a point of integration with Spring. A simple web application can include database connection information in the configuration file, but for larger application it is probably more appropriate for Spring to manage data sources centrally as beans.

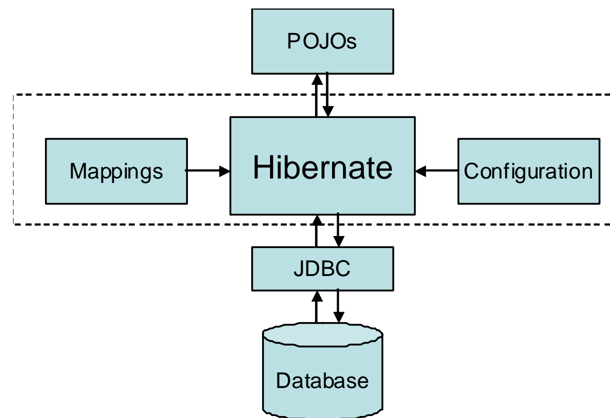

Figure 19 — Hibernate with System Boundary

5.5.2 An example Spring MVC Architecture with Hibernate

The technology stack diagram (Figure 16) shows the range of compatible technologies required for an example web application using Spring MVC, Spring and Hibernate. Connections to the database are expected to go through Hibernate but it is also possible to connect directly to the database using custom code. This example web application combines functionalities provided by the key architectural technologies and services (including Spring MVC, Spring, Hibernate, Tomcat, Ehcache, SiteMesh, and JSTL).

Spring MVC is shown in the example technology stack and technology architecture in order to compare the architecture to other inherently MVC frameworks. It is part of the Spring Framework; in Spring 3.1, you need only specify @EnableWebMvc to enable Spring MVC.

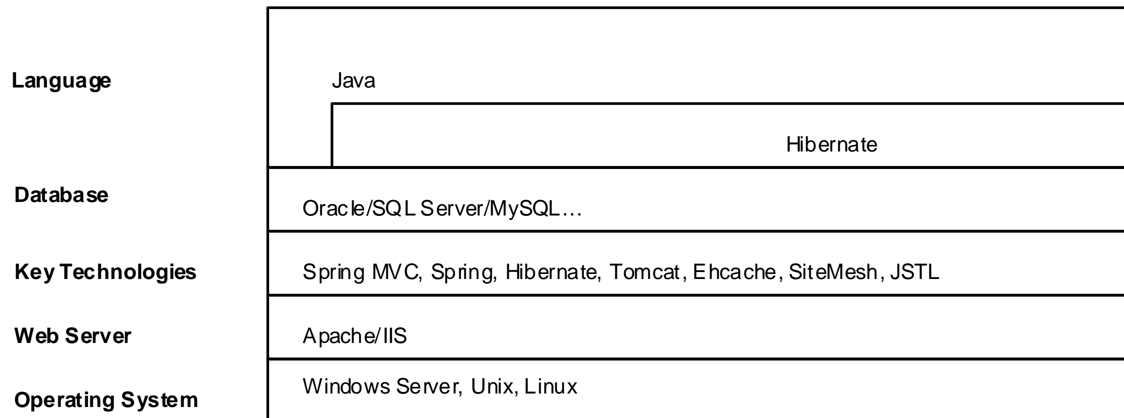

Figure 20 — Spring MVC Hibernate Technology Stack

The architecture diagram in Figure 21 shows a logical view of the technology architecture of an example web application using Spring MVC and Hibernate. The color coding shows the functionalities that are provided by the key architectural technologies and services (including Spring MVC, Spring, Hibernate, Tomcat, Ehcache, SiteMesh, and JSTL) in yellow, and the non-standard, highly customized parts of the application that must be built by the developers in purple. The pre-built functionality provided by a compatible mix of technologies can make it easier and faster to build a fully functioning database-backed web application than coding each function as a custom module. Developers and architects who are experienced with integrating the different technologies can have a significant impact on the success of the project.

The expected method of accessing the database is through Hibernate. However, the purple line in Figure 17 indicates that it is possible for developers to create custom code which accesses the database directly via JDBC.

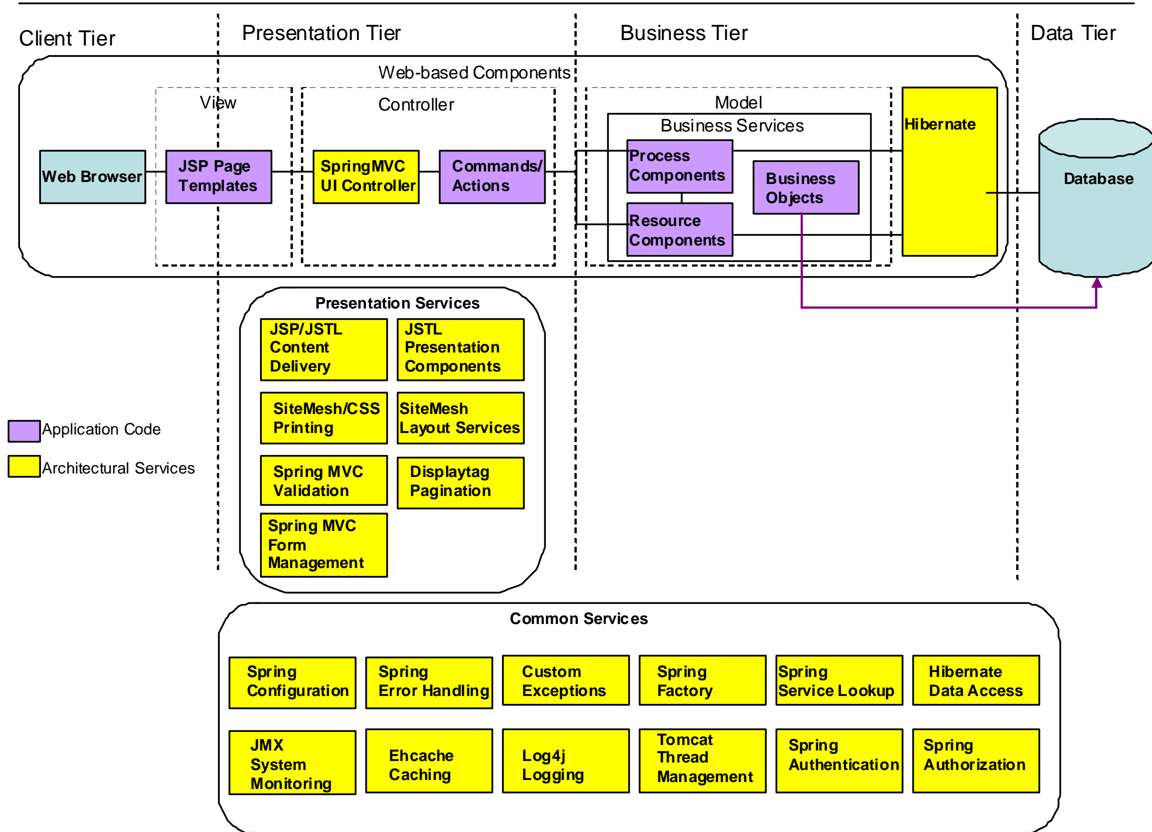

Figure 21 — Spring MVC Hibernate Logical Technology Architecture Diagram

5.5.3 How Does Hibernate Query the Database?

Hibernate Query Language (HQL) is an object-oriented query language that queries persistent objects and their properties, rather than tables and columns [34]. At runtime, Hibernate translates the HQL queries into conventional SQL queries. Hibernate also provides an API that allows for SQL queries to be issued directly.

Here is an example of an unsuccessful SQL injection attack attempt.

a) The web application code specifies the query using HQL with a named parameter:

```
Query safeHQLQuery = session.createQuery("from Members where
lname=:searchName");
          safeHQLQuery.setParameter("lname", searchName);
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS—
```

c) Hibernate manages the user input as a parameterized query.

d) The query will return no rows.

It is important to note that it is possible to write vulnerable HQL , similar to embedded SQL. Here is an example of an HQL-specified query that is vulnerable to an injection attack:.

```
Query unsafeHQLQuery = session.createQuery("from Members
where lname='"+searchName+"'");
```

In the unsafe query above, the user input is not parameterized, but inserted directly into the query string. This makes it vulnerable to any of the attacks described in section 3. Here is an example of a successful SQL injection attack:

a) The web application code specifies the query using HQL with a named parameter:

```
Query unsafeHQLQuery = session.createQuery("from Members
where lname='"+searchName+"'");
```

b) Instead of giving the expected value, a last name of a member of the organization, a malicious user enters the following:

```
smith' UNION SELECT visacardno FROM PAYMENTS—
```

c) The query that the web application will create and execute on the database becomes:

```
SELECT fname, phone, fax, email FROM members
WHERE lname = ‘’;
UNION SELECT visacardno FROM payments;
```

d) The new query will display the information stored in the ‘visacardno’ column for all the rows in the “payments” table. When the results are not restricted by a “where” clause, all results are returned.

```
1234123412341234
3456345634563456
4567456745674567
```

5.5.4 Analysis

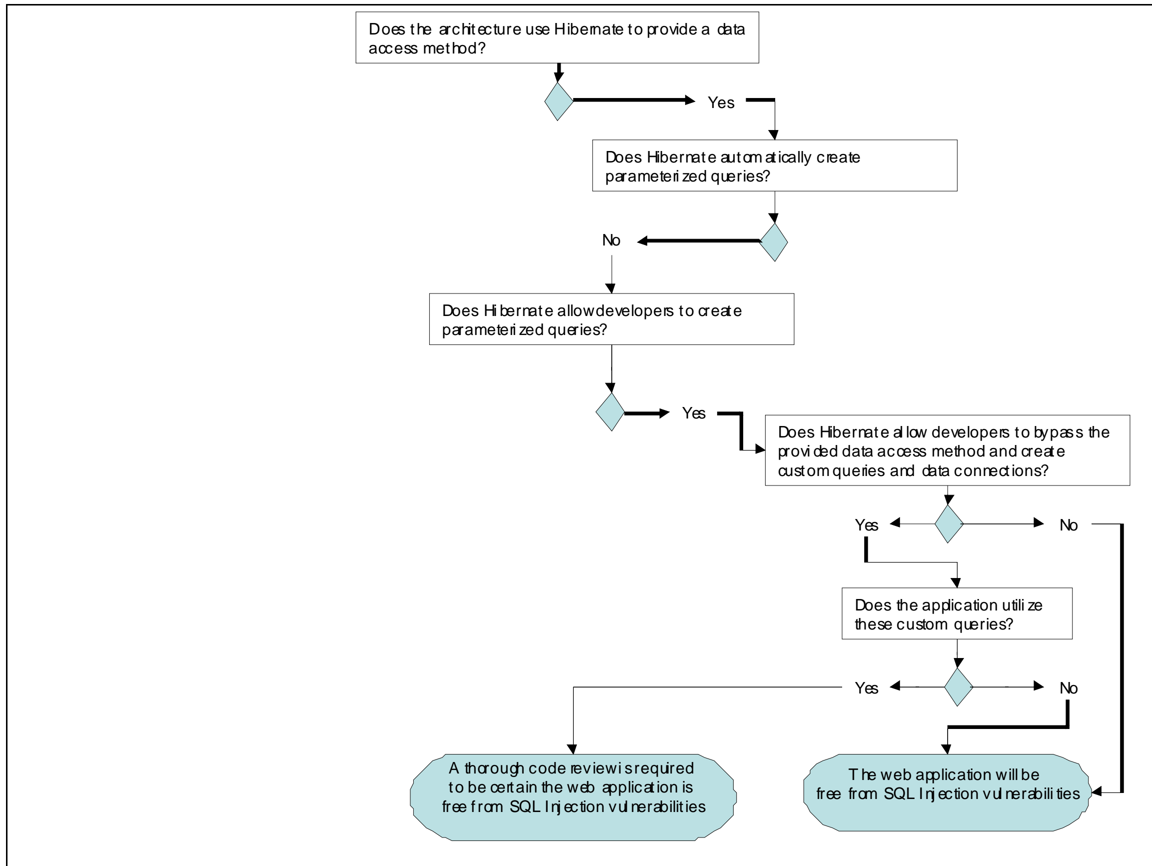

Figure 22 — Hibernate Data Access Method Analysis

Compared to the data access methods provided by other web application frameworks, Hibernate does a very good job of allowing a developer to build an application that is free from SQL injection vulnerabilities using HQL. The documentation is addressed to a highly technical audience of proficient Java developers, and contains no explicit warnings about SQL injection or other security concerns.

Methods on Query are provided for binding values to named parameters, in the format ":name" in the query string [21]. In this way, developers can specify parameterized queries in HQL. It is also possible to create an unsafe query, by concatenating strings rather than using a parameterized query, but this is not discussed in the documentation. As noted in Figure 22, there is also a method of creating custom queries. Hibernate allows developers to specify handwritten SQL, including stored procedures, for all CRUD (Create, Read, Update, Delete) operations.

Because Hibernate is not as easy to use for a novice web application developer, it may not provide as much of a false sense of security that SQL injection vulnerabilities will be handled automatically by the ORM without careful attention from the developer.

5.6 Other PHP ORMs

Many PHP frameworks do not include an ORM (object relation mapping) tool. Researchers are continuing to develop PHP code analyzers such as Ardilla for the Zend framework. The Zend framework is the second most popular framework for PHP applications according to the findings in Appendix 1. In addition there

are several open source projects (in various stages of development) to create an ORM functionality that mimics ActiveRecord, Hibernate, or LINQ. Several of them are listed below:

a) Doctrine – PHP Object Relational Mapper (www.doctrine-project.org) database queries in a proprietary object oriented SQL dialect called Doctrine Query Language (DQL), inspired by Hibernates HQL.

b) phpDataMapper (www.phpdatamapper.com) This project draws inspiration from the Ruby DataMapper project.

c) Pork.dbObject (www.schizofreend.nl/pork.dbobject/) optimized database queries and provides an easy Find() function very loosely based on Ruby on Rails

d) Repose (www.repose-php.org) Repose is an ORM tool for PHP. Similar to Hibernate

e) PHP ActiveRecord (www.derivante.com/…) The PHP ActiveRecord is inspired by Ruby on Rails ActiveRecord.

f) PHPLinq (www.phplinq.NET) PHPLinq is an attempt to port .NET's LINQ (Language-Integrated Query) to PHP.

6. Results

The rubric described in Table 2 and shown in Figure 23 was created to express relative differences between the ideal in protecting from SQL injection vulnerabilities, and less effective methods. Points were awarded for stronger protection methods; a higher score indicates a framework with better protection.

The highest score received by a framework in this analysis was a 10. A framework with a score of 10 automatically defaults to parameterized queries. A framework with this score would be unlikely to have SQL injection vulnerabilities even if a programmer who was inexperienced with the framework created the queries after skimming the documentation enough to build the most obvious command syntax.

Theoretically, a framework could be so secure that it would not allow a programmer to bypass the safe data access method provided. A framework that did this would have been awarded a score of 20, indicating that it was twice as effective as a framework with a score of 10. Unfortunately, there does not appear to be a market for a framework this restrictive, since none of the popular frameworks had this characteristic.

A score of 8 in this analysis represents framework that presents a clearly defined path for developers to properly construct parameterized queries. A novice developer would need to be guided in the proper way to construct safe queries, but it does not entail extra work to make the queries safe.

A score of 5 in this analysis represents a framework that filters user input rather than parameterizing the queries. Automatically filtered user input is significantly less than ideal (awarded half of the points that defaulting to parameterized queries is awarded), but still a significant improvement over filtering the input within the application code (awarded 0 points).

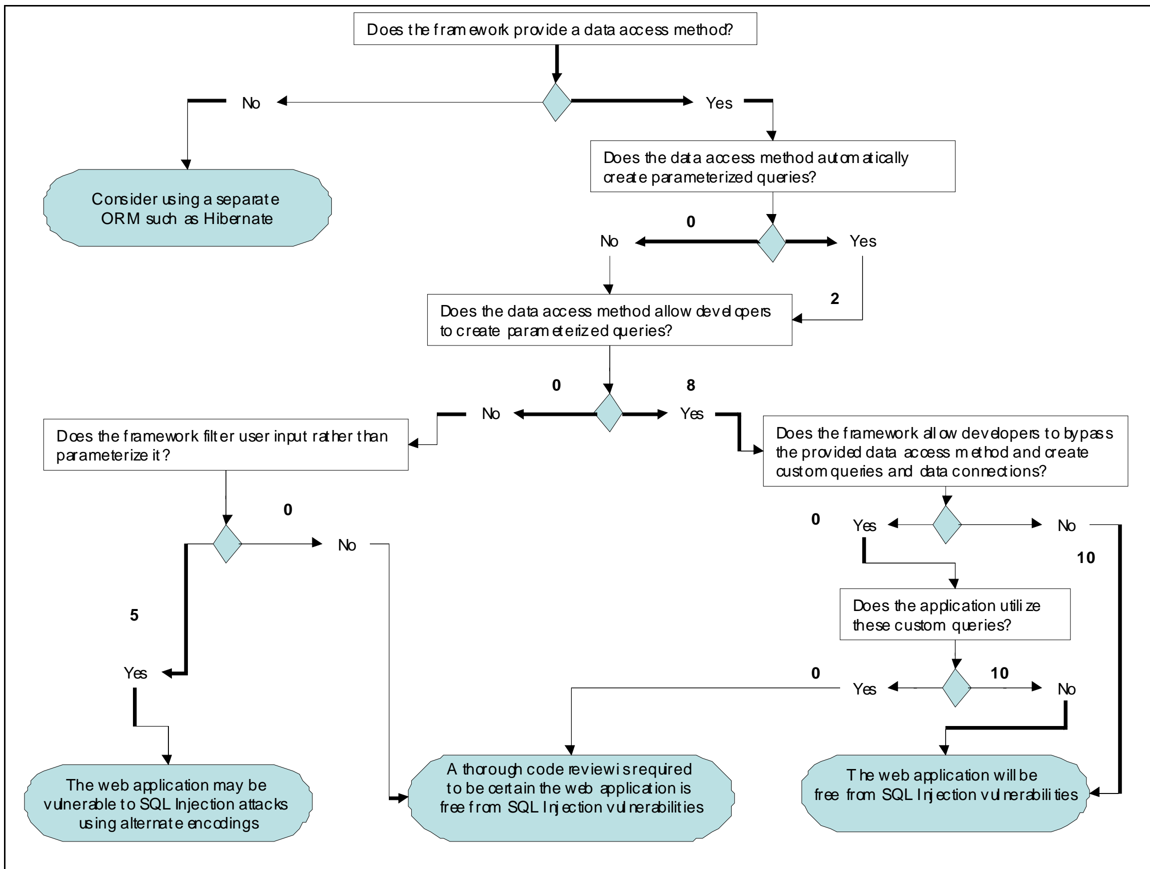

Figure 23 — Weighted Analysis Flow of Web Application Frameworks

Table 2— Rubric for Comparing Frameworks

Criteria	Yes	No
Does the data access method automatically create parameterized queries?	2	0
Does the data access method allow developers to create parameterized queries?	8	0
Does the framework automatically filter user input rather than parameterize it?	5	0
Does the framework allow developers to bypass the provided data access method and create custom queries and data connections?	0	10
Does the application utilize these custom queries?	0	10

With the rubric in place, each framework was analyzed and the results recorded in Table 3. For details of each of the analysis flow diagrams please refer to in the discussion of each framework within Section 5.

Table 3— Framework Comparison

Criteria	LINQ	Ruby on Rails	CodeIgniter	Django	Drupal	Hibernate
Does the data access method automatically create parameterized queries?	2	0	0	0	2	0
Does the data access method allow developers to create parameterized queries?	8	8	0	0	8	8
Does the framework automatically filter user input rather than parameterize it?	0	0	5	5	0	0
Does the framework allow developers to bypass the provided data access method and create custom queries and data connections?	0	0	0	0	0	0
Does the application utilize these custom queries?	N/A	N/A	N/A	N/A	N/A	N/A
Totals	**10**	**8**	**5**	**5**	**10**	**8**

Based on the rubric, the results were clustered in three levels – with two frameworks each scoring 10, 8, and 5. The top scoring data access methods used with web application frameworks are .NET with LINQ and Drupal, which both automatically create parameterized queries. The second tier of scores was awarded to Ruby on Rails and Hibernate. By this analysis, all four of these technologies are considered successful at helping developers create web applications that are free from SQL injection vulnerabilities.

The lowest scoring frameworks were CodeIgniter and Django. These two frameworks did not help developers create injection-proof parameterized queries, but instead provided automated filtering of user input. This is still an improvement over filtering the input in the application code. The framework will help to thwart known SQL injection attack methods, but even with frequent framework updates there will still be vulnerable time windows when innovative attacks are discovered until the appropriate framework response can be implemented.

One of the interesting results found in the analysis of the scoring was the contrast between the different implementations of Active Record in Ruby on Rails and Code Igniter. Even though Ruby on Rails and CodeIgniter both use the Active Record design pattern, they are implemented differently and do not provide the same level of protection against SQL injection vulnerabilities. CodeIgniter does not provide a way to create parameterized queries using Active Record or its other built-in data access methods. While CodeIgniter does offer automated filtering of known malicious user entries, it may not protect against all possible encoding and obfuscation of attacks.

This disadvantage may explain why CodeIgniter earns a significantly lower rank than Drupal or Zend Framework among PHP Frameworks. The drop-off of hits is particularly steep in Dice job postings. It may also be interesting to note that the Zend framework is one of many PHP frameworks that do not provide built-in ORM functionality.

Table 4— Percentage Share of Search Hits Returned for PHP Frameworks by Search Context

Project	Weighted Avg	Search YC		Dice		Amazon	
Drupal	41.50%	1472	40.89%	230	34.64%	148	49.17%
Zend Framework	22.59%	537	14.92%	207	31.17%	73	24.25%
CodeIgniter	8.12%	370	10.28%	38	5.72%	23	7.64%

Another interesting result was the robustness of the oldest technology: Hibernate (2001). A Java/Spring/Hibernate architecture remains a very popular choice as shown in Table 5. Spring clearly dominates in the number of job postings on Dice, which is heavily weighted toward corporate recruiters and hiring managers. There are 3099 job listings for Spring and the next closest is .NET at 633 postings. Even though frameworks like CodeIgniter and Django were created after Hibernate, they did not emulate the choice to use parameterized queries to protect against SQL injection vulnerabilities.

Table 5— Percentage Share of Search Hits Returned for Analyzed Frameworks by Search Context

Language	Project	Weighted Avg.	Search YC		Dice		Amazon	
ASP.NET	ASP.NET MVC	87.83%	504	92.65%	633	88.78%	107	80.45%
Java	Spring	50.68%	494	23.83%	3099	86.98%	145	50.17%
PHP	Drupal	41.50%	1472	40.89%	230	34.64%	148	49.17%
PHP	CodeIgniter	8.12%	370	10.28%	38	5.72%	23	7.64%
Python	Django	70.65%	7324	84.36%	135	77.14%	61	45.86%
Ruby	Ruby on Rails	94.08%	15514	97.34%	628	96.02%	151	87.79%

The least secure frameworks, CodeIgniter and Django, did not do as well in the popularity rankings. CodeIgniter and Django have the fewest job postings on Dice compared to the other frameworks, and the fewest number of books available on Amazon.

Despite this, Django is the second most discussed framework according to Search YC. However, Paul Graham, the founder of YCombinator and a frequent participant in discussions at Hacker News wrote an essay in his book Hackers and Painters[15] in which he praises the Python programming language. Of all the Python frameworks, Django has significantly more job postings on Dice and books available at Amazon.

Table 6— Percentage Share of Search Hits Returned for Python Frameworks by Search Context

Project	Weighted Avg	Search YC		Dice		Amazon	
Django	70.65%	7324	84.36%	135	77.14%	61	45.86%
Pylons	7.37%	551	6.35%	15	8.57%	10	7.52%
Zope 3	7.23%	170	1.96%	6	3.43%	24	18.05%
CherryPy	4.43%	114	1.31%	7	4.00%	12	9.02%

The most secure frameworks by this analysis are Drupal and ASP.NET MVC using LINQ. Both of these frameworks had scores in the middle – a significant but not exceptional number of books available on Amazon, jobs posted at Dice, and discussions on Hacker News.

Table 7— Percentage Share of Search Hits Returned for Most Secure Frameworks by Search Context

Language	Project	Weighted Avg.	Search YC		Dice		Amazon	
ASP.NET	ASP.NET MVC	87.83%	504	92.65%	633	88.78%	107	80.45%
PHP	Drupal	41.50%	1472	40.89%	230	34.64%	148	49.17%

7. Conclusion

Frameworks that automatically escape input are less secure than frameworks that use parameterized queries. A framework that attempts to strip out individual characters but does not parameterize the query may leave the application vulnerable to an injection attack disguised with alternate encoding, or obfuscation. In its discussion of the problem of obfuscation, the IBM X-force® 2010 mid-year trend and risk report [59] explains: "The reason attackers can successfully employ these well known standards to hide their activities, is because many security products cannot interpret every possible encoding/decoding combination and will not detect the attack. This allows for new attack methods that must constantly be reviewed in order to provide detection."

With respect to SQL injection, the framework is in the same position as any other "security product." Using one of the frameworks which automatically escapes user input is much better than attempting to escape the input in the application code. Filtering input is not a trivial undertaking, and popular open-source frameworks are likely to be diligent in their efforts. Such frameworks will provide protection against known attack methods. Frequent framework updates will be required to keep a web application free from new SQL injection attack methods as they arise. Even with due vigilance, there will be time windows of vulnerability between the discovery of an innovative attack method and the updated framework response.

Software development practitioners such as architects, application designers and programmers have many competing considerations when choosing among various popular web application frameworks. Even when a decision is made to start a project (or continue an existing project) with one of the frameworks that is deemed "less safe" by the analysis presented here it is helpful to have a clear understanding about how much protection the framework offers in terms of SQL injection vulnerabilities, and whether there are practices for specifying queries that must be observed in order to create secure applications. This work shows that it is not enough to choose the most popular framework among developers in a particular language and assume that the framework's data access methods are sufficiently safe from SQL injection vulnerabilities.

REFERENCES

[1] Anonymous (2011, June 11). Amazon search: Books › computers & internet. *2011(June 11),* Available: http://www.amazon.com/.

[2] Anonymous (2011, June 11). Dice.com - job search for technology professionals. *2011(June 11),* Available: http://www.dice.com/.

[3] C. Anley, "Advanced SQL Injection In SQL Server Applications," 2002.

[4] C. Anley, "(more) Advanced SQL Injection," 2002.

[5] S. Bandhakavi, P. Bisht, P. Madhusudan and V. N. Venkatakrishnan, "CANDID: Preventing sql injection attacks using dynamic candidate evaluations," in CCS '07: *Proceedings of the 14th ACM Conference on Computer and Communications Security,* 2007, pp. 12-24.

[6] G. Buehrer, B. W. Weide and P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," in *SEM '05: Proceedings of the 5th International Workshop on Software Engineering and Middleware,* 2005, pp. 106-113.

[7] M. Cheng and A. Miller. (2011, June 11). Search Y combinator - an independent project to build a search utility for Y combinator's hacker news. *2011(June 11),* Available: http://searchyc.com/.

[8] L. Constantin. (2010, January 9th, 12:14 GMT). Army website compromised through SQL injection. *Softpedia News 2011(03/26),* Available: http://news.softpedia.com/news/Army-Website-Compromised-Through-SQL-Injection-131649.shtml.

[9] L. Constantin. (2010, November 6th, 10:26 GMT). Hacker claims full compromise of royal navy website. *Softpedia News 2011(03/26),* Available: http://news.softpedia.com/news/Hacker-Claims-Full-Compromise-of-Royal-Navy-Website-165112.shtml.

[10] L. Constantin. (2009, December 22nd, 20:01 GMT). Intel website compromised through SQL injection. *Softpedia News 2011(03/26),* Available: http://news.softpedia.com/news/Intel-Website-Compromised-Through-SQL-Injection-130494.shtml.

[11] Django Software Foundation. (2011, March 23). Django 1.3 documentation. *2011(June 05),* Available: https://docs.djangoproject.com/en/1.3/.

[12] EllisLab Inc. (2011, April 7). CodeIgniter user guide version 2.0.2. *2011(June 05),* Available: http://codeigniter.com/user_guide/database/queries.html.

[13] P. Finnigan. (2004, 03/20/2004). Detecting SQL injection in oracle. *SecurityDocs. Com Security White Papers and Articles (03/31/2008),* Available: http://www.securitydocs.com/library/667.

[14] M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA, USA [download]: Addison-Wesley Longman Publishing Co., Inc, 2002.

[15] P. Graham, *Hackers and Painters: Essays on the Art of Programming.* Sebastopol, CA, USA: O'Reilly \& Associates, Inc, 2004.

[16] M. Gunderloy, P. Naik and X. Noria. (2010, April 7). Ruby on rails guides: Active record query interface. *2011(June 05),* Available: http://guides.rubyonrails.org/active_record_querying.html.

[17] W. G. Halfond, J. Viegas and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," March, 2006.

[18] A. Holovaty and J. Kaplan-Moss, *The Definitive Guide to Django: Web Development done Right.* Apress, 2007.

[19] R. Jennings, *Leverage LINQ in ASP.NET 3.5 Projects.* Indianapolis, IN: Wrox Press, 2008.

[20] S. Joshi. (2005, 09/23/2005). SQL injection attack and defense. *SecurityDocs. Com Security White Papers and Articles (03/31/2008),* Available: http://www.securitydocs.com/library/3587.

[21] G. King, C. Bauer, M. R. Andersen, E. Bernard, S. Ebersole and H. Ferentschik. (2011, April 6). HIBERNATE - relational persistence for idiomatic java: Hibernate reference documentation. *2011(June 05),* Available: http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/.

[22] S. Klein, *Professional LINQ.* Indianapolis, IN: Wrox Press, 2008.

[23] S. Kost. (2007, March 2007). An introduction to SQL injection attacks for oracle developers. *SecurityDocs. Com Security White Papers and Articles (03/31/2008),* Available: http://www.securitydocs.com/library/2481.

[24] N. Kumar, *LINQ Quickly: A Practical Guide to Programming Language Integrated Query with C#.* Birmingham, B27 6PA, UK: Packt Publishing, 2007.

[25] L. Lawson. (2005, 06/06/2005). Introduction to SQL injection. *SecurityDocs. Com Security White Papers and Articles (03/31/2008),* Available: http://www.securitydocs.com/library/3348.

[26] B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium,* 2005, pp. 18.

[27] O. Maor and A. Shulman. (2004, 3/31/2004). Blindfolded SQL injection. *SecurityDocs. Com Security White Papers and Articles (03/31/2008),* Available: http://www.securitydocs.com/library/1196.

[28] F. Marguerie, S. Eichert and J. Wooley, *LINQ in Action.* Greenwich, CT: Manning Publications, 2008.

[29] S. McDonald. 03/24/2004). SQL injection: Modes of attack, defense, and why it matters. *SecurityDocs. Com Security White Papers and Articles (03/31/2008),* Available: http://www.securitydocs.com/library/925.

[30] V. P. Mehta, *Pro LINQ Object Relational Mapping with C# 2008.* Apress, 2008.

[31] J. D. Meier, (2010, October). Web application security frame. *2010(December 17),* Available: http://www.freepatentsonline.com/7818788.html.

[32] Microsoft Corporation, *Improving Web Application Security: Threats and Countermeasures.* Microsoft Press, 2003.

[33] Microsoft Corporation. LINQ (language-intergrated query). *MSDN Library 2011(June 05),* Available: http://msdn.microsoft.com/en-us/library/bb397926.aspx.

[34] D. Minter and J. Linwood, *Beginning Hibernate: From Novice to Professional.* Apress, 2006.

[35] K. K. Mookhey and N. Burghate. 03/17/2004). Detection of SQL injection and cross-site scripting attacks. *(03/31/2008),* Available: http://www.securityfocus.com/infocus/1768.

[36] M. Muthuprasanna, K. Wei and S. Kothari, "Eliminating SQL Injection Attacks - A Transparent Defense Mechanism," *Web Site Evolution, IEEE International Workshop on,* vol. 0, pp. 22-32, 2006.

[37] M. Pelaez. (2010, 2010-08-15). Obfuscated SQL injection attacks. *Internet Storm Center 2011(2/20),* Available: http://isc.sans.edu/diary.html?storyid=9397&.

[38] P. Pialorsi and M. Russo, *Programming Microsoft LINQ.* Redmond, Washington: Microsoft Press, 2008.

[39] T. Pietraszek and C. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Recent Advances in Intrusion Detection*, A. Valdes and D. Zamboni, Eds. Springer Berlin / Heidelberg, 2006, pp. 124-145

[40] B. Prince. (2009, December 22, 3:08 PM). An unpleasant anniversary: 11 years of SQL injection. *EWeek Security Watch 2011(03/26),* Available: http://securitywatch.eweek.com/sql_injection/an_unpleasant_anniversary_eleven_years_of_sql_injection.html.

[41] rain.forest.puppy, "NT Web Technology Vulnerabilities," *Phrack Magazine,* vol. 8, pp. article 08 of 12, Dec 25th. 1998.

[42] N. Rappin, *Professional Ruby on Rails.* John Wiley & Sons (US), 2008.

[43] J. Rattz, *Pro LINQ—Language Integrated Query in C# 2008.* Berkeley, CA: Apress, 2007.

[44] F. S. Rietta, "Application layer intrusion detection for SQL injection," *Proceedings of the 44th Annual Southeast Regional Conference,* Melbourne, Florida, 2006, pp. 531-536.

[45] S. Sanderson, *Pro ASP.NET MVC 2 Framework, Second Edition.* Apress, 2010.

[46] J. Schmitt. (2007, May 24). Eliminate SQL injection attacks painlessly with LINQ. . *DevX. Com 2008(10/15/2008),* Available: http://www.devx.com/dotnet/Article/34653.

[47] K. Spett, "SQL Injection, Are Your Web Applications Vulnerable?" *SPI Labs White Paper,* 2004.

[48] K. Spett. 10/26/2004). Blind SQL injection. (03/31/2008), .

[49] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications,” *SIGPLAN Not.,* vol. 41, pp. 372-382, January, 2006.

[50] The Drupal Association. (2011, May 25). Drupal API reference. *2011(June 05),* Available: http://api.drupal.org/api/drupal/includes--database--database.inc/group/database/7.

[51] D. Thomas, D. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland and A. Schwarz, *Agile Web Development with Rails.* Pragmatic Bookshelf, 2006.

[52] Treeface. (2010, October 12). Prevent SQL injections in CodeIgniter. *Stack Overflow* Available: http://stackoverflow.com/questions/3917831/prevent-sql-injections-in-codeigniter.

[53] D. Upton, *CodeIgniter for Rapid PHP Application Development: Improve Your PHP Coding Productivity with the Free Compact Open-Source MVC CodeIgniter Framework!* Packt Publishing, 2007.

[54] F. Valeur, D. Mutz and G. Vigna, "A learning-based approach to the detection of SQL attacks," 2005.

[55] J. K. VanDyk, *Pro Drupal Development, Second Edition.* Apress, 2008.

[56] G. Wassermann and Z. Su, "An analysis framework for security in web applications," In *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS) 2004,* 2004, pp. 70-78.

[57] K. Wei, M. Muthuprasanna and S. Kothari, "Preventing SQL injection attacks in stored procedures," in *Proceedings of the Australian Software Engineering Conference,* 2006, pp. 191-198.

[58] D. Wichers. (2011, May 27). SQL injection prevention cheat sheet. *OWASP 2011(June 29),* Available: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.

[59] B. Williams, C. Hagemann, J. Thamm, F. Licitra, H. Moss, J. Larimer, L. Horacek, M. Noske, M. Wallis, M. Waidner, M. Alvarez, M. Warfield, R. Iffert, R. Srinivasan, R. Freeman, R. McNulty, S. Moore, T. Cross and W. McKelvey. (2010, August). IBM X-force® 2010 mid-year trend and risk report. IBM Security Solutions.

APPENDIX A: DETERMINING THE PERTINENT FRAMEWORKS

The frameworks were grouped according the development language: ASP.NET, Java, PHP, Python and Ruby The name of each framework in the five groups was used as a search term in the three different search engines: Amazon.com [1], Dice.com [2], and SearchYC [7]. The number of hits for each search term was recorded, and the percentage share of the total hits for the group was calculated. For example, Drupal received 1472 hits on SearchYC. This represented 40.89% of the 3600 total hits for all PHP frameworks discussed on Hacker News.

A weighted average was then calculated. The percentage share for Search YC was weighted at 40%, and each of the percentage shares for Dice and Amazon were weighted at 30%.

Table 8— Percentage Share of Search Hits Returned for All Frameworks by Search Context

ASP.NET							
Project	**Weighted Avg**	**Search YC**		**Dice**		**Amazon**	
ASP.NET MVC	87.83%	504	92.65%	633	88.78%	107	80.45%
DotNetNuke	8.36%	17	3.13%	35	4.91%	25	18.80%
Csla	1.65%	7	1.29%	27	3.79%	0	0.00%
BFC	0.79%	1	0.18%	17	2.38%	0	0.00%
MonoRail	0.78%	10	1.84%	1	0.14%	0	0.00%
OpenRasta	0.59%	5	0.92%	0	0.00%	1	0.75%
Total		**544**		**713**		**133**	
Java							
Project	**Weighted Avg**	**Search YC**		**Dice**		**Amazon**	
Spring	50.68%	494	23.83%	3099	86.98%	145	50.17%
Google Web Toolkit	19.32%	863	41.63%	95	2.67%	18	6.23%
Apache Struts	8.27%	235	11.34%	37	1.04%	33	11.42%
WebObjects	3.22%	107	5.16%	14	0.39%	10	3.46%
Apache Wicket	2.91%	93	4.49%	21	0.59%	9	3.11%
JavaServer Faces	1.81%	1	0.05%	28	0.79%	15	5.19%
Apache Tapestry	1.41%	46	2.22%	0	0.00%	5	1.73%
IceFaces	1.35%	4	0.19%	16	0.45%	11	3.81%
Oracle ADF	1.32%	2	0.10%	54	1.52%	8	2.77%
Apache Click	1.22%	57	2.75%	2	0.06%	1	0.35%
JBoss Seam	1.22%	12	0.58%	80	2.25%	3	1.04%
OpenLaszlo	1.13%	47	2.27%	2	0.06%	2	0.69%
Stripes	1.06%	28	1.35%	13	0.36%	4	1.38%
WebWork	1.05%	29	1.40%	21	0.59%	3	1.04%
Vaadin(IT Mill Toolkit)	0.72%	31	1.50%	2	0.06%	1	0.35%
Apache Cocoon	0.57%	2	0.10%	1	0.03%	5	1.73%
AppFuse	0.51%	5	0.24%	0	0.00%	4	1.38%
Hamlets	0.45%	2	0.10%	0	0.00%	4	1.38%
Aranea	0.45%	0	0.00%	53	1.49%	0	0.00%
ZK	0.29%	1	0.05%	7	0.20%	2	0.69%

Java (continued)							
Project	**Weighted Avg**	**Search YC**	**Dice**	**Amazon**			
Wavemaker	0.25%	7	0.34%	1	0.03%	1	0.35%
OpenXava	0.23%	1	0.05%	0	0.00%	2	0.69%
Eclipse RAP	0.14%	2	0.10%	0	0.00%	1	0.35%
ItsNat	0.12%	1	0.05%	0	0.00%	1	0.35%
FormEngine	0.10%	0	0.00%	0	0.00%	1	0.35%
Sling	0.08%	0	0.00%	9	0.25%	0	0.00%
SmartClient	0.06%	2	0.10%	3	0.08%	0	0.00%
Shale	0.03%	0	0.00%	3	0.08%	0	0.00%
ThinWire	0.02%	1	0.05%	0	0.00%	0	0.00%
Play!	0.02%	0	0.00%	2	0.06%	0	0.00%
Jspx-bay	0.00%	0	0.00%	0	0.00%	0	0.00%
JVx WebUI	0.00%	0	0.00%	0	0.00%	0	0.00%
ManyDesigns Portofino	0.00%	0	0.00%	0	0.00%	0	0.00%
RIFE	0.00%	0	0.00%	0	0.00%	0	0.00%
ztemplates	0.00%	0	0.00%	0	0.00%	0	0.00%
Totals		**2073**		**3563**		**289**	

PHP							
Project	**Weighted Avg**	**Search YC**		**Dice**		**Amazon**	
Drupal	41.50%	1472	40.89%	230	34.64%	148	49.17%
Zend Framework	22.59%	537	14.92%	207	31.17%	73	24.25%
CodeIgniter	8.12%	370	10.28%	38	5.72%	23	7.64%
Joomla!	7.49%	327	9.08%	50	7.53%	16	5.32%
CakePHP	7.47%	310	8.61%	56	8.43%	15	4.98%
symfony	6.18%	253	7.03%	57	8.58%	8	2.66%
Kohana	3.33%	234	6.50%	14	2.11%	1	0.33%
Yii	1.47%	61	1.69%	2	0.30%	7	2.33%
Lithium	0.36%	11	0.31%	3	0.45%	1	0.33%
Modx	0.29%	0	0.00%	2	0.30%	2	0.66%
e107	0.27%	6	0.17%	0	0.00%	2	0.66%
Horde	0.23%	3	0.08%	0	0.00%	2	0.66%
Qcodo	0.19%	8	0.22%	0	0.00%	1	0.33%
Seagull	0.14%	0	0.00%	1	0.15%	1	0.33%
Solar	0.10%	1	0.03%	2	0.30%	0	0.00%
PPI Framework	0.10%	0	0.00%	0	0.00%	1	0.33%
Alloy	0.06%	1	0.03%	1	0.15%	0	0.00%
Rain Framework	0.05%	0	0.00%	1	0.15%	0	0.00%
Zeta Components	0.03%	3	0.08%	0	0.00%	0	0.00%
DooPHP	0.02%	2	0.06%	0	0.00%	0	0.00%
Sapphire	0.01%	1	0.03%	0	0.00%	0	0.00%
Kajona	0.00%	0	0.00%	0	0.00%	0	0.00%
Quick PHP	0.00%	0	0.00%	0	0.00%	0	0.00%
Thin PHP Framework	0.00%	0	0.00%	0	0.00%	0	0.00%
Zext	0.00%	0	0.00%	0	0.00%	0	0.00%
Totals		**3600**		**664**		**301**	

Python							
Project	**Weighted Avg**	**Search YC**		**Dice**		**Amazon**	
Django	70.65%	7324	84.36%	135	77.14%	61	45.86%
Pylons	7.37%	551	6.35%	15	8.57%	10	7.52%
Zope 3	7.23%	170	1.96%	6	3.43%	24	18.05%
CherryPy	4.43%	114	1.31%	7	4.00%	12	9.02%
TurboGears	3.57%	125	1.44%	3	1.71%	11	8.27%
Grok	3.45%	87	1.00%	2	1.14%	12	9.02%
web2py	1.58%	134	1.54%	3	1.71%	2	1.50%
Pyjamas	0.53%	78	0.90%	1	0.57%	0	0.00%
Pyramid	0.51%	25	0.29%	1	0.57%	1	0.75%
Flask	0.32%	70	0.81%	0	0.00%	0	0.00%
BlueBream	0.17%	0	0.00%	1	0.57%	0	0.00%
Webware	0.17%	0	0.00%	1	0.57%	0	0.00%
Nagare	0.02%	4	0.05%	0	0.00%	0	0.00%
CubicWeb	0.00%	0	0.00%	0	0.00%	0	0.00%
Totals		8682		175		133	

Ruby							
Project	**Weighted Avg**	**Search YC**		**Dice**		**Amazon**	
Ruby on Rails	94.08%	15514	97.34%	628	96.02%	151	87.79%
Sinatra	4.35%	339	2.13%	23	3.52%	14	8.14%
Merb	0.84%	1	0.01%	3	0.46%	4	2.33%
Camping	0.37%	10	0.06%	0	0.00%	2	1.16%
Nitro	0.24%	25	0.16%	0	0.00%	1	0.58%
Ramaze	0.12%	49	0.31%	0	0.00%	0	0.00%
Totals		**15938**		**654**		**172**	

APPENDIX B: A THOROUGH CODE REVIEW

The analysis of many of the frameworks has suggested that a thorough code review is needed to determine whether the application is free from SQL injection vulnerabilities. A good place to begin is by identifying all of the vulnerable points in the application and its execution architecture. The purpose of an architecture and design security review described by Microsoft Press [32] is to analyze the architecture and design in order to identify:

a) all points of entry / control of the system

b) data flow

c) trust boundaries

d) the potential threats and level of acceptable risk

Another tool to identify areas of concern is a detailed checklist. The nine categories of vulnerability defined for Web applications by the Web Application Security Frame [31] can be used to ensure no key areas are missed during the review. SQL Injection vulnerabilities are considered under the "Input Validation" category as shown in Table 9.

Table 9— Checklist for Security Review of Architecture Design for the Input Validation Category

Input Validation	The application should constrain, reject, and sanitize all of the input it receives. Constraining input is the best approach because validating data for known valid types, patterns, and ranges is much easier than validating data by looking for known bad characters
Entry Points	The design should identify all entry points of the application in sufficient detail to track what happens to individual input fields, including: Web page input, input to components, and input from databases
Trust Boundaries	Input validation is mandatory if the input is passed from sources that are not trusted
Web page input	Do not consider the end user as a trusted source of data. Validate regular and hidden form fields, query strings, and cookies
Input from the database	Validate this form of input, especially if other applications write to the database. Make no assumptions about how thorough the input validation of the other application is.
Centralized Approach	Ensure that validation rules are performed consistently
What does the application do with input	Check what your application does with its input because different types of processing can lead to various types of vulnerabilities. For example, if you use input in SQL queries your application is potentially vulnerable to SQL injection
SQL Injection	Pay close attention to any input field that you use to form a SQL database query. Check that these fields are suitably validated for type, format, length, and range. Also check how the queries are generated. If you use parameterized stored procedures, input parameters are treated as literals and are not treated as executable code. This is effective risk mitigation

Once all of the vulnerable points in the design have been identified, special care can be taken to examine those sections of the application code to enforce coding standards and secure data access practices. If all of the queries that take input from the end user are created as parameterized queries, the application is deemed free from SQL injection vulnerabilities.