SQL Injections Attacks and Aversion Techniques

Amit Mahale, Gokulanand Viswanath, Kunal Narsinghani, Radhika Dharurkar
{amahale1, gokul1, kunal2, radhika1}@umbc.edu
Department of Computer Science, UMBC

Abstract:

Ever since Web Applications came into existence in this internet world and started using SQL databases for data management, there evolved a devastating vulnerability called 'SQL injection'. In this attack, attackers use SQL queries as a weapon to reach backend data using power and flexibility of supporting database and operating system functionality. A malicious user thus gains illegal access of the remote machines through the web applications vulnerability. In this paper, we survey the various SQL injection vulnerabilities and techniques for its detection and prevention. We also propose a SQL injection detection system which is more effective in the current scenario.

1. Introduction:

1.1 What is a SQL Injection?

It is the vulnerability which exposes the database to an attacker by providing the ability to influence the SQL queries passed by an application to the back-end database.

1.2 Architecture of web applications

A database-driven Web application commonly has three tiers: a presentation tier (Web browser or rendering engine), a logic tier (a programming language, such as C#, ASP, .NET, PHP, JSP, etc.), and a storage tier (a database such as Microsoft SQL Server, MySQL, Oracle, etc.) [3].

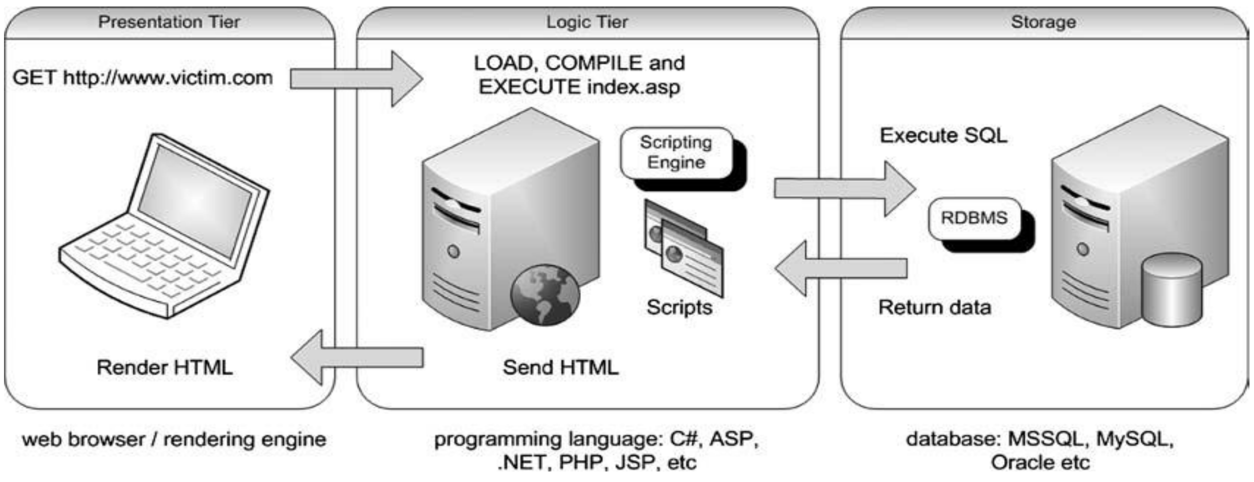

Presentation Tier
Logic Tier
Storage
GET http://www.victim.com
LOAD, COMPILE and EXECUTE index.asp
Scripting Engine
Scripts
Execute SQL
RDBMS
Return data
Render HTML
Send HTML
web browser / rendering engine
programming language: C#, ASP, .NET, PHP, JSP, etc
database: MSSQL, MySQL, Oracle etc

Figure 1: Web application - 3 tier architecture

SQL Injection occurs at the logic tier, where the attacker maliciously inserts commands to be executed by the storage tier.

An example of SQL injection is as follows:

In the case of an online retail store, using get parameters, products which cost $100 can be fetched using the URL: http://www.victim.com/products.php?val=100

The scripting engine builds and executes the query as

*SELECT * FROM Products WHERE val = '100.00' ORDER BY ProductDescription;*

The malicious attacker can inject SQL commands by concatenating them to the val parameter as http://www.victim.com/products.php?val=100' OR '1'='1. This time, the following query is executed,

*SELECT * FROM Products WHERE val = '100.00' OR '1'='1' ORDER BY ProductDescription;*

Since the second clause of the query is always true (1=1), the attacker has access to all the products in the system, and the system is vulnerable. A similar method can be employed by an attacker to expose content management systems (CMS), which need user credentials prior to accessing any of their functionalities.

The gravity of SQL Injection can be determined by the fact that the Computer Fraud and Abuse Act of 1986 [7] as well as the USA PATRIOT Act of 2001 include prosecution clauses for it.

1.3 Core causes of SQL Injection

- SQL injection occurs when the Web application developer does not ensure that values received from a Web form, cookie, input parameter are validated before passing them to SQL queries that will be executed on a database server. If an attacker can control the input that is sent to an SQL query and manipulate that input so that the data is interpreted as code instead of as data, the attacker may be able to execute code on the back-end database.
- Another cause is the use of black lists in place of white lists. While black lists include a default allow policy for characters, white lists are based on a default deny policy, and only allow specified characters for a given input and deny anything else.
- Besides, attackers often circumvent sequential validation which developers assume is often the case in applications.
- Instances where error handling mechanisms built into the application are verbose and reveal additional information to the attacker. These include database dumps and error codes. Verbose error messages provide the attacker clues regarding security holes in the application.
- Insecure database configurations which employ default users e.g.: sa, sys & scott. As a result, server services should always be run as an unprivileged user. The ideal method would be to provide a least-privilege model for the application's database access.
- Freely available metadata on Database systems – This vulnerability boils down to the access of metadata available under popular database products such as Oracle and SQL server. SQL

Server includes this under the view INFORMATION_SCHEMA and Oracle in ALL_TABLES [3].

E.g. Oracle statement to enumerate all accessible tables for the current user
SELECT OWNER, TABLE_NAME FROM ALL_TABLES ORDER BY TABLE_NAME;

MS SQL statement to enumerate all accessible tables using the catalog views
SELECT name FROM sys.tables;

1.4 How can SQL Injections be avoided?

Application developers often build SQL queries dynamically using string building at runtime. Instead of using dynamic string building techniques to generate queries on the fly, developers can use secure coding practices using parameterized queries. Parameters can then be passed to these queries at runtime. Besides, these parameters cannot be interpreted as commands to execute and thus code cannot be injected.
An example of this method is a dynamically built SQL string statement in .NET,

*query = "SELECT * FROM table WHERE field = '" + request.getParameter("input") + "'";*

Most databases today include defense mechanisms against SI by disallowing queries that include escape characters blank space ( ), double pipe (||), comma (,), period (.), (*/), and double-quote characters (“).

2. Related Work:

2.1 General Exploitation technique

As we have seen before, the basic idea behind SQL injection is to run the SQL query which was not intended to run by a programmer. This technique heavily relies on the logical operations like AND, OR, UNION etc. It is common for SQL injection vulnerabilities to occur in SELECT statements, which do not modify data. However we cannot rule out attacks which use INSERT, UPDATE, and DELETE, to modify the contents of the database. The backend details like database type, version and table details should be concealed to avoid easy access to hackers to inject SQL attacks [3].

Steps followed by attacker:

1. The primary step followed by a malicious user is to identify the backend database by forcing it to return an error, thus the web programmer should mask this bug and return a generic error without revealing any details of the database and the error that was caused by the end user.
2. After identifying the type of database, the next target is to extract data from the database; this is done by appending data to existing query. The value NULL is used as the wildcard as it is accepted for all the data types and GROUP BY helps to find the exact number of columns to inject, by following this procedure data can be extracted row wise. A simple technique to prevent such attacks would be to avoid concatenated queries.
3. Once the intricate details of the database are exposed, then the hacker can follow a hierarchical approach: enumerating the databases, then the tables of each database, then the

columns of each table, and then finally the data of each column. If the remote database is huge, it is usually unnecessary to extract it in its entirety; a quick look at the table names is usually enough to spot where the interesting data is. Thus using code names instead of logical names like account, depositor would help to reduce the attacks.
4. Another type of attack is of escalating privileges, this vulnerability is caused if the system does not have the latest security fixes. Once the admin account is compromised it can lead the way to grab the password hashes. As most people have a tendency to reuse passwords, this can be a major security concern and also risk the organizations reputation in maintaining users' data.

2.2 Classification of SQL Injection Attacks (SQLIAs)

[4] Defines two important characteristics of SQLIAs for describing attacks: injection mechanism and attack intent.

2.2.1 Injection Mechanisms – Malicious SQL statements can be introduced into a vulnerable application using many different input mechanisms - injection through user input, injection through cookies, injection through server variables and second order injections.

2.2.2 Attack Intent – Attacks can be classified based on the attacker's motive, as - identifying injectable parameters, performing database finger-printing, determining database schema, extracting data, adding or modifying data, performing denial of service, evading detection, bypassing authentication, executing remote commands, performing privilege escalation.

2.2.3 SQLIA Types – The different kinds of attacks are not performed in isolation; usually many of them are used together or sequentially, depending on the specific goals of the attacker. There are countless variations of each attack type – Tautologies, Illegal/Logically Incorrect Queries**,** Union Query, Piggy Backed Queries, Stored Procedures, Inference and Alternate Encodings**.**[4]

2.3 Blindfolded SQL Injection

The common solution to the problem discussed in the previous section is by suppressing the detailed error messages. Since most documents describing SQL Injection rely on gathering information through the error messages. Though this reduces the chances of SQL Injection, it is not a fool proof solution as there exist attacks which can be carried out even without detailed error messages which explicitly mention the intricate details of the backend. This section provides insight on this type of attack known as "Blindfolded SQL Injection"[5]

Steps followed in Blindfold SQL injections:
1. The first step in "Blindfolded SQL Injection" is to recognize the errors. Errors are broadly categorized into two categories the first type of error is that generated by the Web server as the result of some exception in the code. If untouched, these exceptions yield the familiar 500: Internal Server Error. Normally, injection of bad SQL syntax (unclosed quotes, for instance), should cause the application to return this type of error, although other errors may lead to such an exception. The error can be suppressed by directing to the index page or the previous page that the user navigated, or to a generic error message which does not provide any information about the error. The second type of error is generated by the program code i.e. the application. The application expects certain invalid cases, and generates specific

tailored errors for them. Although normally these types of errors come as part of a valid (200 OK) response, they may also be replaced with redirecting to previous page or other means of concealing, similar to the generic errors like Internal Server Error. An attacker with the intention to perform Blindfolded SQL Injection would therefore try to query a few invalid requests, to learn the applications strategy to handles errors, and what could be expected of it when a SQL error occurs. With that knowledge of the application at hand, the attacker can now proceed to the second part of the attack, which is locating errors that are a result of a manipulated input. For that step, normal SQL Injection testing techniques are applied, such as adding SQL keywords (OR, AND, etc.), discussed earlier. Each parameter is individually tested and the response is closely examined to determine whether an error occurred. Using an intercepting proxy or any other tool of choice, it is easy to identify redirects and other supposedly hidden errors. Each parameter that returns an error is suspicious, as it may be vulnerable to SQL Injection.

2. The next task is to identify the SQL injection vulnerable parameters which can be exploited. This can be done by having a basic understanding of the type of data types that SQL supports. Techniques to test whether the field uses Numeric of string data are used to find ways of exploiting these parameters.

3. The final task is to perform the injection, once the vulnerability is detected by the attacker, the next step will be trying to exploit it. For that step, the attacker using all the knowledge gained by previous steps, must be able to generate valid syntax, identify the specific Database Server, and build the required query. Attackers often have a desire to be able to perform a UNION SELECT injection since successfully performing a UNION SELECT injection gives access to all tables in the system which otherwise may not be accessible. Performing a UNION SELECT statement is not that simple as it requires knowledge of the number of fields in the query as well as the type of each field.

2.4 Prevention of SQLIAs

2.4.1 Defensive Coding Practices - The root cause of SQLIAs is insufficient input validation. The straightforward solution is to apply suitable defensive coding practices, such as
- Input type checking
- Encoding of inputs - attackers sometimes use meta-characters that trick the SQL parser into interpreting user input as SQL tokens. This needs to be handled by using functions to treat the entire string as normal string.
- Positive pattern matching - establish input validation routines that identify good input as opposed to bad input.
- Identification of all input sources - Must check all possible sources of input to an application.

2.4.2 Detection and Prevention Techniques – Various techniques have been proposed for assisting developers to reduce the shortcomings of defensive coding during the development of an application, such as - Black Box Testing, Static Code Checkers, Combined Static and Dynamic Analysis, Taint Based Approaches, New Query Development Paradigms, Proxy Filters, Platform-level, Intrusion Detection Systems, Instruction Set.[4]

2.5 Reviewing code for SQL Injection

The most effective way of finding potential areas of SQL injection in an application is to review the source code. Source codes can be analyzed either manually or can be subjected to automated tools to identify potential areas of SQL injection [3].

There are two main methods of manually analyzing a source code – static code analysis, in which source code is analyzed without executing the code, and dynamic code analysis, in which code is analyzed during execution of the application. Reviewing code is mainly aimed at finding taint-style vulnerabilities. Tainted data is data that has been received from an untrusted source. Known sanitization techniques or functions for validating input data can be used to untaint tainted data. Tainted data causes security problems at vulnerable points in the source code and such points are known as sinks. Once a sink is identified, it is very obvious that a SQL injection is vulnerable at that point.

Thus, it is highly necessary to ensure that values given to a sink undergo validation before being passed to SQL queries, which are run on a database. However, reviewing a source code for sinks is a tedious process. All dependencies need to be mapped and data flows have to be traced to identify tainted and untainted inputs. The process of reviewing the source code has to start with identifying functions that build and execute SQL statements. Following this, we should identify entry points for user-controlled data that is being passed to these functions and, finally, trace the user-controlled data through the application's execution flow to ascertain whether the data is tainted when it reaches the sink.

To ease the task of reviewing code manually, we automate the process and thereby create automation tools. Complex scripts or programs to grab various patterns in source code and link them together need to be built. However, performing an effective source code review and identifying all potential SQL injection vulnerabilities, needs the ability to recognize dangerous coding behaviors, such as codes that incorporate dynamic string-building techniques. Adding to this, each programming language offers a number of different ways to construct and execute SQL statements.

2.6 Exploiting the Operating System

In SQL injection, the attacker uses SQL queries as the weapon to reach database using the power and flexibility of supporting database and operating system functionality available to the database. We have seen the database support in earlier sections. Now, we will concentrate on the operating system support for SQL injection. Almost any command execution can be translated fairly easily to remote file reading through many of the same channels used through the database. Oracle offers various possibilities to read files from the underlying operating system. Most of them require the ability to run PL/SQL code. The goal of attacker would be to be able to load a binary as UNSAFE. To do this, however, requires that our binary be signed during development and that our key be trusted to the database. This would seem like too much of a mission to overcome through injection, but we are afforded a way out, since we can simply set the database to "Trustworthy" to bypass this limitation [3].

Many of the attacks that are aimed at compromising the underlying operating system require that the SQL user be running with elevated privileges. In the early days, such elevation was not necessary, when the principle of least privilege was less understood and when every application connected to the back-end database with full db-sys admin privileges. Therefore, most automated SQL injection toolkits provide the ability to identify the current user's privilege level as well as multiple methods for possibly elevating him from a standard database user to a database super user.

Applications may have set up many input filters which normally block SQL attacks which are mostly using SQL keywords, specific characters (quotation marks) and whitespaces. Attackers find different techniques to handle unusual features of applications to deliver a successful attack. Following are some of the techniques [3]:

- Using dynamic query execution: Many databases allow dynamic execution of SQL queries by passing a string containing a SQL query, which could be to circumvent the filters. An attacker can construct individual characters using the *CHAR* function (*CHR* in Oracle) using their ASCII character code. He can construct strings in this way without using any quotation mark characters or using the *CHAR* function to place strings (such as *'admin'*). On the SQL platform one can also use a single hexadecimal number which represents the string's ASCII character code for a string.
- Case Validation: Assuming keyword-blocking filters to be naïve, an attacker can bypass them by varying the case of characters in the attack string.
- Using SQL Comments: The attacker can use inline comment sequences to create snippets of SQL.
- URL Encoding: This technique is normally used to defeat many kinds of input filters. They also use double-encoding since web applications decode use input more than once.
- Truncation Attacks: These effective attacks are mostly used on passwords.
- Second order SQL injection: Since first order SQL injection is captured by good filters, attackers use this technique which involves more than one http requests to an application.
- Other techniques like Null byte attack, nesting stripped expressions

3. Proposed System Architecture:

Our SQL Injection Detection System addresses intrusions against database systems. The operation of the system is described as follows:

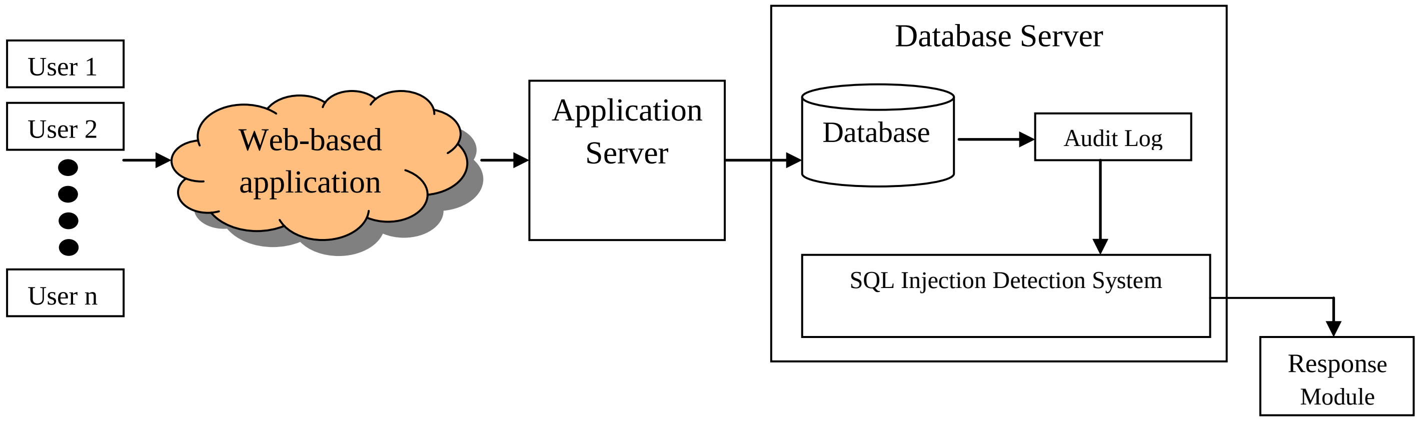
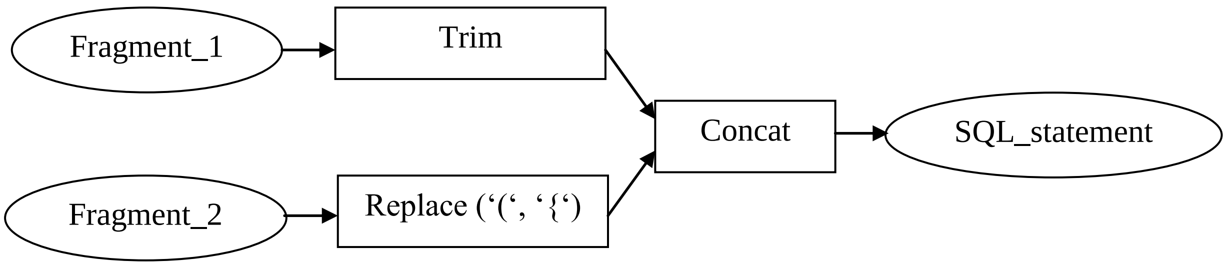


Figure 2: SQL Injection Detection System Architecture - Overview

The user issues a service request to the application server through a web-based application. The SQL query statements are deployed by the application server and issued to the database server. The SQL statements are received by the audit and channeled to the SQL Injection detection module. In the module, the received SQL statements are matched with the set of SQL injection's signatures. If an intrusion had occurred, it is channeled to the Respond Module [6].

The SQL Injection Detection module works on a two stage process for detecting, preventing and reporting SQL Injection incidents. In the first stage we use a static analysis / program analysis techniques which represents the SQL-queries as Finite State Automata (FSA) and views them as a SQL-graph. In the next stage we deploy the runtime validation mechanism which does not require any code modification. A simple web server patch is sufficient for its functioning. It uses Java String Analysis library which is an inter-procedural data-flow analysis that abstracts the control flow of the program and represents the semantics of string manipulation operations on string variables as a flow graph. They define certain hotspots in the target program where the application code issues SQL queries to the underlying database. The string analysis uses the SOOT Framework [6] to parse a class file and produce inter-procedural control-flow graphs to give Non-Deterministic Finite State Automaton (NDFA) that expresses all the possible values a particular string can assume, using single character transitions in the automaton. Therefore, any user input would be compared against this SQL-FSM and any change in the SQL-FSM structure would indicate a possible Injection attack. A SQL-graph helps to optimize number of queries that need to be put under the scanner during runtime to ensure the validity of dynamically generated queries. Links from one SQL query to another are represented by dependency in SQL-graphs [1].



SQL-statement = fragment_1.Trim() + Fragment_2.Replace('(','{');

Figure 3: String Analysis

In runtime validation, SQL queries with embedded user input are compared to SQL-FSM. The technique uses a Verification Table which is computed for the different SQL queries indicating whether they can be allowed to pass through or not before being sent to the database server. This checking process results in degraded Quality of Service (QoS) to the end-users. The directed dependency in the SQL-graph tells us which SQL queries are supersets of which other SQL queries in the SQL-graph. It would suffice to check only those SQL queries that are supersets of other queries and thus implicitly check the other queries encompassed by it. Thus the process can filter out all those SQL queries that have no directed dependency edges coming into them and verify only the validity of the SQL-FSMs corresponding to those SQL queries. Following figure describes the way the defense mechanism works.

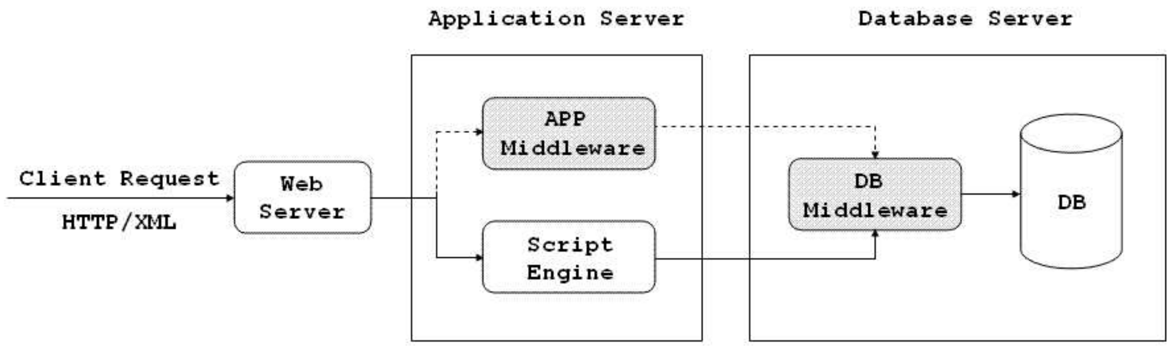


Figure 4: Detailed SQL Injection Detection System Architecture

Normal flow when a XML/HTML client request is received has been shown by solid line. In modified architecture, along with this path, request is transferred to application middleware which sends verification table to database middleware. Database middleware also receives dynamically generated queries from script engine. Trace Table (TT) can also be added with both the Script Engine and the Application Middleware, so that the different SQL-FSMs and the various branches that were processed in both the blocks are stored in order.

The same concept can be applied to a stored procedure parser, so as to verify the input for presence of injection. A control flow graph of the stored procedure in the system with procedure dependencies indicated by arrows is also used. EXEC(@SQL) commands expect user input and are hence identified for analysis. These dynamically generated statements are then compared against the original statement structure using FSA (finite state automaton) [2].

4. Implementation:

Following tasks can be performed on the code to prevent SQLIAs [3].

1. Alternatives to dynamic string building.
2. Strategies to perform input validation
3. Encoding of the output
4. Canonicalization of data.
5. Design level consideration for secure applications

4.1 Parameterized statements

Instead of dynamic string building, most programming languages provide API’s which allow parameterized statements to be executed. These are safer than dynamic string building since the attacker cannot alter the logic tier code, and can be used to optimize the query.

4.2 Input validation

The mechanisms for input validation could be white-list validation (positive validation) or black-list validation (negative validation). White-list validation involves accepting only known valid inputs. Regex validations of the input fall in this category of validation. The other checks include data type (positive or negative), max length, data range, and content. Blacklist validation also makes use of Regular expressions, though with a list of characters or strings to disallow.

An example of input validation performed in ASP.NET is the use of the RegularExpressionValidator and CustomValidator controls. To improve the user experience at the client side, client validation functions are used in the script.

4.3 Output encoding

Strings created for use in dynamic SQL treat the single quote as a terminator, and it thus needs to be encoded. If they are not encoded, the substring following the single quote can be interpreted as a SQL command, and is this vulnerable to SI.
This is done as follows:
sql = sql.Replace(“ ’ ”, “ ’ ’ ”);

4.4 Canonicalization

This is the process of reducing the input to a standard or simple form; for instance the single quote could be encoded as %27 in a URL. The input validation approach thus needs to include canonicalization. Normalization can be performed in C# using the Normalize method of the string class as follows:
normalized = input.Normalize(NormalizationForm.FormKC);

4.5 Design level consideration for secure applications

These include use of stored procedures, which allow configuring access control at the database level. Stored procedures restrict the attacker from accessing sensitive database information. Other methods include an additional layer of abstraction which handles database access and ensures that all the database calls are made using parameterized statements.

Considerations would be to avoid obvious names for critical objects in the database such as passwords and sensitive columns. A last method is setting up database honey pots which alert the administrators via e-mail when the sensitive fields of a database are under attack.

In the implementation phase of our project, the emphasis has been on the security aspects of the Oracle database from a .NET perspective. Oracle data provider for .NET (ODP.NET) provides some built in feature which allows securing applications built on the .NET platform.

Auditing logs while reaping the benefits of connection pooling is harder, since all users login using the Oracle credentials, not the actual user id. We propose using proxy authentication thus facilitating per-user authentication to the database server and demonstrating single sign on to the database using Windows authentication. Using a proxy connection we create a lightweight session which destroyed once the connection is returned to the pool. Another method to utilize the benefits of connection pooling without authorizing the user is to pass the client ID. A demonstration of this approach can be seen in our work. Finally, we demonstrate one method to avoid SQL injection attacks using parameterized queries. Scenarios demonstrating the different kinds of SQL Injection attacks and means to avert them, though not exhaustive, are also covered.

5. Conclusion:

In this paper, we have covered the various techniques of performing SQL Injections and the techniques to detect and prevent these attacks. The core causes of Injection attacks and the common exploitation techniques used by attackers have also been discussed. We then follow up with an in depth classification of the type of these attacks and then hash out countermeasure strategies. The system architecture of the SQL Injection detection system is then presented, which consists of the misuse detection module which employs static analysis and runtime validation.

In order to demonstrate, we have implemented a subset of the techniques presented in this paper using oracle as the backend and developing web interface using C#.NET technology.

In the future, we would like to collect the performance data of our system and also train the misuse detection module using machine learning technique and compare its performance with the existing system.

References:

[1] M. Muthuprasanna, Ke Wei, Suraj Kothari, "Eliminating SQL Injection Attacks - A Transparent Defense Mechanism," wse, pp.22-32, Eighth IEEE International Symposium on Web Site Evolution (WSE'06), 2006

[2] M. Muthuprasanna, Ke Wei, Suraj Kothari, “Preventing SQL Injection Attacks in Stored Procedures”, In Proceedings of the Australian Software Engineering Conference, 2006

[3] Justin Clarke et al., SQL Injection Attacks and Defense

[4] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In Proc. of the Intern. Symposium on Secure Software Engineering
(ISSSE 2006), Mar. 2006.

[5] Blindfolded SQL Injection, A white paper by Imperva, A Data Security Organization.

[6] Aziah Asmawi, Zailani Mohamed Sidek, Shukor Abd Razak, “System Architecture for SQL Injection and Insider Misuse Detection System for DBMS”, 2008

[7] www.cio.energy.gov/documents/ComputerFraud-AbuseAct.pdf