Securing Asp.Net Applications Against Sql Injections Attacks

S. Sayeed, A.A.A. Al-habsi, I. Yusof, J. Hossen, S.M.A. Kalaiarasi

Faculty of Information Science and Technology (FIST), Multimedia University (MMU), Malaysia.
Faculty of Engineering and Technology (FET), Multimedia University, Malaysia.

Abstract: Most of the companies nowadays uses websites to run their operations as well as store confidential data; therefore securing those websites is very vital. Unfortunately this is easier said than done. Any individual can use some programs and tools to launch an attack. These attacks cost companies millions of dollars every year. The increasing number of potential security vulnerabilities coupled with the ease of launching an attack makes it very challenging to secure a web application. In this paper we try to analyze some of the common attacks and what can be done to prevent them by identifying the potential vulnerabilities of the application and how the attacker will take advantage of them then what can be done to prevent that.

Key words: ASP.NET, SQL Injection, Blind SQL Injection.

INTRODUCTION

Structured Query Language is the standard language to interact with a database. SQL allows the user or to define and manipulate data in the database through specific statements. The user can create a database where he can create tables (Knowledge Base – Indiana University, 2012). These tables are used to store data. SQL allows the user to insert, update and delete data as well as build up queries which will return some specific fields from the database based on the condition “WHERE”. Many web applications currently use database servers to run their websites. MS SQL, My SQL and Oracle are the most commonly used database servers. The fact that those database servers hold vital information makes them a target for computer attackers. SQL injections attack can be caused by having a poorly written code that has no control over the user input. The system can’t rely on the user to provide a valid user input. In addition, SQL injections occurs usually when the user input is directly used to build SQL statements, in the case the user input will be treated as an executable code rather a literal value. The primary process in this attack is inserting malicious code in the user-input variables which are used to complete the SQL query that will be passed to the server for execution. In this paper, we proposed a new model for defence against SQL injection.

This paper is organized into five sections. Section I gives a brief introduction about the Structured Query Language which might be used by computer attackers. In section II described about the background and related work, section III described about the proposed model for defence against SQL injection. Section IV discussed the expermetal outcomes and the final conclusions are drawn in Section V.

Background and Related Work:

SQL injection is a type of attack where harmful codes are inserted into dynamic strings. These strings are passed to the database server as part of the SQL query where it is going to be executed (How To: Protect From SQL Injection in ASP.NET- Microsoft, 2012).

The basic idea of this attack revolves around using the user input to directly construct SQL statements. A simple attack can follow these simple steps:

1- The end user application asks the user to enter his credentials.

2- The end user enters a malicious code.

3- The system executes undesirable action.

Generally there are two types of this attack: First order SQL injections and second order SQL injections. The first order attack is where the user inserts malicious code within SQL statements. This type is the most common one. In second order attack, the data inside the database is used to construct dynamic SQL statements which will result in having dynamic SQL injections (SANS- Reading Room, 2009).
A simple demonstration of the SQL injection in the .NET framework can be seen in the following example:

```csharp
SqlCommand cmd = new SqlCommand("SELECT * FROM Customer WHERE username = "+ user.Text + " AND password = " + password.Text + ", con);
```

Here the SQL query includes dynamic strings which are going to be filled by the user through the web application. A malicious user may want to tamper with the SQL query to bypass the authentication process by inserting a command like:

```
' or 1=1--
```

in the login or password field This will result in bypassing the authentication check. This is simply because 1=1 is always true. The resultant code will be something like this:

```
"SELECT * FROM Customer WHERE username = " or 1=1--" AND password = " + password.Text + ",".
```

Now “OR” statement can be true if one or both part of the statements are true. The single quotation mark closes the first statement while it is clear the second statement is true 1=1. In addition, the comments characters (--) ensures the rest of the statement will be treated as a comment and therefore will not be evaluated.

The reason behind this result is the fact that the user-input was treated as an executable code. A malicious user can cause a bigger damage by using a query such as:

```csharp
SELECT * FROM Customer
WHERE username = "+ user.Text + ";
DELETE FROM Customer
```

This query will clearly have undesirable result of deleting all the customer information from the database.
Blind SQL Injections:
Blind SQL injections are basically the same as the normal SQL injections, except that in the case of blind attack, the attacker won’t be looking for clear error messages. Instead the attacker will perform a series of yes/no tests to get information about the database. The most common SQL query used by any web application is the WHERE clause which is used to retrieve a specific data from a table or more in the database. An attacker can steal information by asking several yes/no questions to the database using SQL queries (Blind SQL Injections-SPI Labs, 2003).

Example of Blind SQL Injection:
Many websites use SQL statement to retrieve data from the database.

A random website URL will look like:

```
```

Most likely the underlying SQL statement will be like:

```
SELECT author, title, body FROM articles WHERE article_ID=4;
```

The database server will return the article which has ID number 4. Now to test if this website is vulnerable to blind SQL injection attack another condition will be added in the URL.

```
www.abc123.com/articles.aspx?ArticleID=4 AND 1=1
```

Now if the database server responds by retrieving article 4, then this means that the user input is treated as an executable code and therefore we can conclude that this website is not secure against SQL injections. The “WHERE” condition can give an attacker a clear answer of true or false. If the “WHERE” condition returned a record then this mean the statement is true while an error message shows that the statement is false. Such behavior can enable the attacker to gain a lot of information about the database by asking different types of questions. Now if the attacker for example wants to know if the current user is “dbo”. The attacker will modify the URL in this way:

```
www.abc123.com/articles.aspx?ArticleID=4 AND USER_NAME()='dbo'
```

Again if the database server responds by retrieving article 4, then the current user is in fact “dbo”, while an error message will indicate such a used doesn’t exist.

Usually the first thing the attacker will try to find out is the name of the first table in the database. The process is relatively long however as the attacker needs to perform a series of tests for each character in the table name string. This can be done by trying to

Proposed Model for Defence Against SQL Injection:
The proposed system must contains a method to validate the user input. Validating user input can be done easily in the .Net framework using client side or server validation or even both. Another proposed method of validation is using the XML Schema. Another important element of the proposed system is having parameterized variables in the code. This will ensure the attacker can’t run a script againsts the database.

User-input Validation:
The key component of preventing this type of attacks is validating the user input. SQL injections can only occur when there is no control over the user input. Validating user input can be done in two ways: client side validation and server side validation. Using both methods is the ideal way of validating user inputs. Client side validation can be more users friendly however; it is not sufficient to control the user input since an attacker can easily bypass the client side validation. This is where the server side validation can be useful to restrict any illegal inputs. Validation in general can be done by testing the type, length and format of all the user inputs. For example rejecting input characters such as: “;”, “-” can help in avoid these type of attacks (Table 1).
Fig. 2: Proposed model for Defence against SQL injection.

Table 1:

<table>
<thead>
<tr>
<th>Input character</th>
<th>Why it should be filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>To insert new malicious command</td>
</tr>
<tr>
<td>,</td>
<td>Character data string delimiter.</td>
</tr>
<tr>
<td>--</td>
<td>Continue the attack even the SQL syntax is incorrect.</td>
</tr>
<tr>
<td>/* ... */</td>
<td>Comments are not evaluated by the server</td>
</tr>
</tbody>
</table>

**XML Schema:**

Validation base on the XML Schema can be used to control user input. The XML Schema can be used as Security Policy Description Language. For each input page a schema file will be created to describe what type of constraint we want to implement (Li et al. 2009).

A few lines of codes are needed to generate an XML schema file for various types of input constraints. To specify a range for any given parameter let’s say we have a parameter “abc” which must be an integer between 0 and 50 then the schema file will be like this:

```xml
<xs:simpleType name="abcType">
   <xs:restriction base="xs:int">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
   </xs:restriction>
</xs:simpleType>
```

Regular expressions are also available in the XML schema file where the input values can be constrained:

```xml
<xs:simpleType name="IDType">
   <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z]\d{9}"/>
   </xs:restriction>
</xs:simpleType>
```
XML schema file can filter the symbols that may appear in SQL injections and cross-site Scripting. In order to do that the following schema file can be used:

```xml
<xs:simpleType name="nameType">
  <xs:restriction base="xs:string">
    <xs:pattern value="[^'";<>%]"/>
  </xs:restriction>
</xs:simpleType>
```

**Executable Codes:**
Another important issue in tackling SQL injections is preventing the user input from being treated as an executable code. This can be done by using parameters where the user input will be treated as a literal value instead of executable code. Parameters can also be used to enforce type and length Checks (How To: Protect From SQL Injection in ASP.NET- Microsoft, 2012).

The code below shows how the user-input can be parameterized:

```sql
SELECT * FROM Customer WHERE( username = @username)AND (password = @password);
```
```csharp
cmd.Parameters.AddWithValue("@username", user.Text);
cmd.Parameters.AddWithValue("@password",password.Text)
```

In this case even if a user attempted to input a malicious code in the input field it will be treated as a literal value and therefore the code won't be executed.

**Experimental Results and Discussion:**
The experiment will involve the attempt of inserting a simple SQL injection in the username field in an attempt to bypass the login authentication.

![Fig. 3: SQL injection attack.](image)

![Fig. 4: Attacker bypassed login.](image)

Firstly the experiment will be carried out when the code uses the user input directly to build the SQL statements (i.e. dynamic strings).

![Welcome ' or 1=1--](image)

Follow the links on the right side to be able to interact with our system. You can seek more information under “New Information Link.”

It is clear that the attacker easily bypasses the authentication required by the website.

Second situation the code will be changed so that whatever the user keys in will be treated as literal values. This can be done by using parameterized variables as mentioned earlier.
Fig. 5: Failed SQL injection attack.

Here the attacker isn’t able to bypass the authentication by using this type of attack. The difference between the 2 experiments was the code that had been used to handle the authentication procedure. In the first experiment, dynamic string was used to take the user-input and generate the SQL statement.

```csharp
SqlCommand cmd = new SqlCommand("SELECT * FROM Customer WHERE username = "+ user.Text +" AND password = " + password.Text + "", con);
```

The major flow of this method is that it allows the attacker to insert a malicious code into the login field. While in the second experiment the attacker can’t bypass the authentication because the malicious that has been inserted will not be executed and will be treated as a literal value.

```csharp
SELECT * FROM Customer WHERE( username = @username)AND (password = @password)");
```

```
cmd.Parameters.AddWithValue (@username, user.Text);
cmd.Parameters.AddWithValue (@password, password.Text)
```

This code is a bit longer but provides much more security and prevents this type of attack.

**Advanced Attacks – SQL Injection Worms:**

A type of automated SQL injection attack appeared in 2008. This type of attack combines the devastating effects of worms as well as SQL injections. Worms are known to be independent malware that can propagate a network quickly. The basic idea of this attack is to inject a JavaScript code into web applications (SANS-Reading Room, 2009). Unlike normal worms the real target of the attack isn’t the direct websites which are infected but the visitors of the infected websites. The SQL injection is only used to transmit the malicious ware to the websites. When a user visits one of those infected websites his PC will be infected as well. Like any regular worms attack, it is very difficult to stop the worm from spreading once it starts propagating and that’s what makes this type of attack very dangerous and has the potential to be used on much larger scale.

**Conclusion:**

According to OWASP, injections are ranked as number 1 in the top 10 security vulnerabilities (OWASP Top 10 Application Security Risks, 2010). SQL injections in particular have affected a large number of systems worldwide. SQL injections attack can cause very severe damage to companies and industries by compromising their databases. However defending against this type of attack is not very difficult. SQL injection vulnerability happens because of a flaw in programming a system. So what causes SQL injection vulnerability might be a poor written code that didn’t consider the risk of internet attacks. Writing a safe code that validate user inputs and filter out suspicious characters can help in eliminating the threat of SQL injection attacks.

**REFERENCES**


