SQL Injection Attack

Introduction

Security in software applications is an ever more important topic. In this article, I discuss various aspects of SQL Injection attacks, what to look for in your code, and how to secure it against SQL Injection attacks. Although the technologies used here are SQL Server 2000 and the .NET Framework, the general ideas presented apply to any modern data driven application framework, which makes attacks potentially possible on any type of application that depends on that framework.

What is a SQL Injection Attack?

A SQL Injection attack is a form of attack that comes from user input that has not been checked to see that it is valid. The objective is to fool the database system into running malicious code that will reveal sensitive information or otherwise compromise the server.

There are two main types of attacks. First-order attacks are when the attacker receives the desired result immediately, either by direct response from the application they are interacting with or some other response mechanism, such as email. Second-order attacks are when the attacker injects some data that will reside in the database, but the payload will not be immediately activated. I will discuss each in more detail later in this article.

An example of what an attacker might do

In the following example, assume that a web site is being used to mount an attack on the database. If you think about a typical SQL statement, you might think of something like:

```
SELECT ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductName LIKE 'G%'
```

The objective of the attacker is to inject their own SQL into the statement that the application will use to query the database. If, for instance, the above query was generated from a search feature on a web site, then they user may have inserted the "G" as their query. If the server side code then inserts the user input directly into the SQL statement, it might look like this:

```
string sql = "SELECT ProductName, QuantityPerUnit, UnitPrice "+
    "FROM Products " +
    "WHERE ProductName LIKE '"+this.search.Text+"%';
SqlDataAdapter da = new SqlDataAdapter(sql, DbCommand);
da.Fill(productDataSet);
```

This is all fine if the data is valid, but what if the user types something unexpected? What happens if the user types:

```
' UNION SELECT name, type, id FROM sysobjects;--
```

Note the initial apostrophe; it closes the opening quote in the original SQL statement. Also, note the two dashes at the end; that starts a comment, which means that anything left in the original SQL statement is ignored.

Now, when the attacker views the page that was meant to list the products the user has searched for, they get a list of all the names of all the objects in the database and the type of object that they are. From this list, the attacker can see that there is a table called *Users*. If they take note of the `id` for the *Users* table, they could then inject the following:

```
' UNION SELECT name, '', length FROM syscolumns
WHERE id = 1845581613;--
```

This would give them a list of the column names in the *Users* table. Now they have enough information to get access to a list of users, passwords, and if they have admin privileges on the web site.

```
' UNION SELECT UserName, Password, IsAdmin FROM Users;--
```

Assume that there is a table called *Users* which has columns called `UserName` and `Password`, it is possible to union that with the original query and the results will be interpreted as if the `UserName` was the name of the product and the `Password` was the quantity per unit. Finally, because the attacker discovered that there is a `IsAdmin` column, they are likely to retrieve the information in that too.

Locking down

Security is something that needs to be tackled on many levels because a chain is only as strong as its weakest link. When a user interacts with a piece of software, there are many links in the chain; if the user is malicious, he could attempt to attack these links to find the weak point and attempt to break the system at that point. With this in mind, it is important that the developer does not become complacent about the security of the system because one security measure is put in place, or a set of security measures are in place on only one part of the system.

An intranet website that uses Windows authentication (it takes the user's existing credentials based on who they are logged in as) and is sitting inside the corporate network and unavailable to Internet users may give the impression that only authorised users can access the intranet web application. However, it is possible for an authenticated user to gain unauthorised access if the security is not taken much beyond that level. Some statistics support the suggestion that most security breaches are insider jobs rather than people attacking the system from outside.

With this in mind, it is important that even if the application permits only valid data through that has been carefully verified and cleaned up, other security measures are

put in place. This is especially important between application layers where there may be an increased opportunity for spoofing of requests or results.

For example, if a web application were to request that the user choose a date, then it would be normal that the values for the date are checked in some JavaScript function on the web page before any data was posted back to the server. This improves the user experience by reducing the wait between lots of server requests. However, the value needs to be validated again on the server as it is possible to spoof the request with a deliberately crafted invalid date.

Encrypting data

Starting from the proposition that somehow an attacker has managed to break through all other defenses, what information is so sensitive that it needs to remain a secret? Candidates for encryption include user log in details or financial information such as credit card details.

For items such as passwords, the user's password can be stored as a "salted hash". What happens is that when a user creates a password, a randomly generated "salt" value is created by the application and appended to the password, and the password-and-salt are then passed through a one way encryption routine, such as found in the .NET Framework's helper class `FormsAuthentication` (HashPasswordForStoringInConfigFile method). The result is a salted hash which is stored in the database along with the clear text salt string.

The value of a salted hash is such that a dictionary attack is not going to work as each dictionary would have to be rebuilt appending the various salt values and recomputing the hash values for each item. While it is still possible to determine the password by brute force, the use of the salt (even though it is known) greatly slows down the process. The second advantage of the salt is that it masks any situations where two independent users happen to use the same password, as the salted hash value for each user would be different if given different salt values.

Least Privilege - Database account

Running an application that connects to the database using the database's administrator account has the potential for an attacker to perform almost limitless commands with the database. Anything an administrator can do, so can an attacker.

Using the example application above, an attacker could inject the following to discover the contents of the hard disk(s) on the server.

The first command is used to create a temporary store on the database and fill it with some data. The following injected code will create a table with the same structure as the result set of the extended stored procedure that will be called. It then populates the table with the results of the extended stored procedure.

```
'; CREATE TABLE haxor(name varchar(255), mb_free int);
INSERT INTO haxor  EXEC master..xp_fixeddrives;--
```

A second injection attack has to take place in order to get the data out again.

```
' UNION SELECT name, cast((mb_free) as varchar(10)), 1.0 FROM haxor;--
```

This returns the name of the disks with the available capacity in megabytes. Now that the drive letters of the disks are known, a new injection attack can take place in order to find out what is on those disks.

```
'; DROP TABLE haxor;CREATE TABLE haxor(line varchar(255) null);
INSERT INTO haxor EXEC master..xp_cmdshell 'dir /s c:\';--
```

And again, a second injection attack is used to get the data out again.

```
' UNION SELECT line, '', 1.0 FROM haxor;--
```

`xp_cmdshell`, by default, is only executable by a user with the `sysadmin` privilege, such as `sa`, and `CREATE TABLE` is only available to `sysadmin`, `db_dbowner` or `db_dlladmin` users. It is therefore important to run the application with the least privileges that are necessary in order to perform the necessary functions of the application.

Least Privilege - Process account

When an instance of SQL Server is installed on a computer, it creates a service that runs in the background and processes the commands from applications that are connected to it. By default, this service is installed to use the Local System account. This is the most powerful account on a Windows machine, it is even more powerful than the Administrator account.

If an attacker has an opportunity to break out of the confines of SQL Server itself, such as through the extended procedure `xp_cmdshell`, then they could gain unrestricted access to the machine that the SQL Server is on.

Microsoft recommends that during the installation of SQL Server, the service is given a domain account which has the permissions set to only the necessary resources. That way, an attacker is confined by the permission set needed to run SQL Server.

Cleaning and Validating input

In many applications, the developer has side-stepped the potential use of the apostrophe as a way to get access to the system by performing a string replace on the input given by the user. This is useful for valid reasons, such as being able to enter surnames such as "O'Brian" or "D'Arcy", and so the developer may not even realise that they have partly defeated a SQL injection attack. For example:

```
string surname = this.surnameTb.Text.Replace("'", "''");
string sql = "Update Users SET Surname='"+surname+"' "+
    "WHERE id="+userID;
```

All of the previous injection attack examples would cease to work given a scenario like this.

However, many applications need the user to enter numbers and these don't need to have the apostrophes escaped like a text string. If an application allows the user to review their orders by year, the application may execute some SQL like this:

```
SELECT * FROM Orders WHERE DATEPART(YEAR, OrderDate) = 1996
```

And in order for the application to execute it, the C# code to build the SQL command might look like this:

```
string sql = "SELECT * FROM Orders WHERE DATEPART(YEAR, OrderDate) = "+
    this.orderYearTb.Text);
```

It becomes easy to inject code into the database again. All the attackers need to do in this instance is start their attack with a number, then they inject the code they want to run. Like this:

```
0; DELETE FROM Orders WHERE ID = 'competitor';--
```

It is therefore imperative that the input from the user is checked to determine that it really is a number, and in the valid range. For instance:

```
string stringValue = orderYearTb.Text;
Regex re = new Regex(@"\D");
Match m = re.Match(someTextBox.Text);
if (m.Success)
{
    // This is NOT a number, do error processing.
}
else
{
    int intValue = int.Parse(stringValue);
    if ((intValue < 1990) || (intValue > DateTime.Now.Year))
    {
        // This is out of range, do error processing.
    }
}
```

Second-Order Attacks

A second-order attack is one where the data lies dormant in the database until some future event occurs. It often happens because once data is in the database, it is often thought of as being clean and is not checked again. However, the data is frequently used in queries where it can still cause harm.

Consider an application that permits the users to set up some favourite search criteria. When the user defines the search parameters, the application escapes out all the apostrophes so that a first-order attack cannot occur when the data for the favourite is inserted into the database. However, when the user comes to perform the search, the data is taken from the database and used to form a second query which then performs the actual search. It is this second query which is the victim of the attack.

For example. If the user types the following as the search criteria:

```
'; DELETE Orders;--
```

The application takes this input and escapes out apostrophe so that the final SQL statement might look like this:

```
INSERT Favourites (UserID, FriendlyName, Criteria)
VALUES(123, 'My Attack', ''';DELETE Orders;--')
```

which is entered into the database without problems. However, when the user selects their favourite search, the data is retrieved to the application, which forms a new SQL command and executes that. For example, the C# code might look like:

```
// Get the valid user name and friendly name of the favourite
int uid = this.GetUserID();
string friendlyName = this.GetFriendlyName();

// Create the SQL statement to retrieve the search criteria
string sql = string.Format("SELECT Criteria FROM Favourites "+
    "WHERE UserID={0} AND FriendlyName='{1}'",
    uid, friendlyName);
SqlCommand cmd = new SqlCommand(sql, this.Connection);
string criteria = cmd.ExecuteScalar();

// Do the search
sql = string.Format("SELECT * FROM Products WHERE ProductName = '{0}'",
    criteria);
SqlDataAdapter da = new SqlDataAdapter(sql, this.Connection);
da.Fill(this.productDataSet);
```

The second query to the database, when fully expanded, now looks like this:

```
SELECT * FROM Products WHERE ProductName = ''; DELETE Orders;--
```

It will return no results for the expected query, but the company has just lost all of their orders.

Parameterised Queries

SQL Server, like many database systems, supports a concept called parameterised queries. This is where the SQL Command uses a parameter instead of injecting the values directly into the command. The particular second-order attack above would not have been possible if parameterised queries had been used.

Where the application developer would have constructed a `SqlCommand` object like this:

```
string cmdText=string.Format("SELECT * FROM Customers "+
    "WHERE Country='{0}'", countryName);
SqlCommand cmd = new SqlCommand(cmdText, conn);
```

A parameterised query would look like this:

```
string commandText = "SELECT * FROM Customers "+
    "WHERE Country=@CountryName";
SqlCommand cmd = new SqlCommand(commandText, conn);
```

```
cmd.Parameters.Add("@CountryName",countryName);
```

The value is replaced by a placeholder, the parameter, and then the parameter's value is added to the `Parameters` collection on the command.

While many second-order attacks can be prevented by using parameters, they can only be used in places were a parameter is permitted in the SQL statement. The application may return a variable sized result set based on user preference. The SQL statement would include the `TOP` keyword in order to limit the result set, however, in SQL Server 2000, `TOP` can only accept literal values so the application would have to inject that value into the SQL command to obtain that functionality. For example:

```
string sql = string.Format("SELECT TOP {0} * FROM Products", numResults);
```

Using Stored Procedures

Stored Procedures add an extra layer of abstraction in to the design of a software system. This means that, so long as the interface on the stored procedure stays the same, the underlying table structure can change with no noticeable consequence to the application that is using the database. This layer of abstraction also helps put up an extra barrier to potential attackers. If access to the data in SQL Server is only ever permitted via stored procedures, then permission does not need to be explicitly set on any of the tables. Therefore, none of the tables should ever need to be exposed directly to outside applications. For an outside application to read or modify the database, it must go through stored procedures. Even though some stored procedures, if used incorrectly, could potentially damage the database, anything that can reduce the attack surface is beneficial.

Stored procedures can be written to validate any input that is sent to them to ensure the integrity of the data beyond the simple constraints otherwise available on the tables. Parameters can be checked for valid ranges. Information can be cross checked with data in other tables.

For example, consider a database that has the user details for a website, this includes the user name and password. It is important that an attacker is unable to get a list of passwords or even one password. The stored procedures are designed so that a password can be passed in, but it will never put a password in any result set. The stored procedures for registering and authenticating a user for the website might be:

- `RegisterUser`
- `VerifyCredentials`
- `ChangePassword`

`RegisterUser` takes the user name and password as parameters (possibly along with other information that is necessary for registering on the website) and returns the `UserID`.

`VerifyCredentials` would be used for logging into the site by accepting the user name and the password. If there is a match the `UserID` is returned, if not then a `NULL` value.

`ChangePassword` would take the `UserID`, the old password and the new password. If the `UserID` and the password match, the password can be changed. A value that indicates success or failure is returned.

The above example shows that the password is always contained in the database and is never exposed.

Stored Procedure Caveat

While stored procedures seem to be a wonderful panacea against injection attacks, this is not necessarily the case. As mentioned above, it is important to validate data to check that it is correct and it is a definite benefit of stored procedures that they can do this; however, it is doubly important to validate data if the stored procedure is going to use `EXEC(some_string)` where `some_string` is built up from data and string literals to form a new command.

For instance, if the stored procedure is to modify the data model of the database, such as creating a table, the code may be written as follows:

```
CREATE PROCEDURE dbo.CreateUserTable
    @userName sysname
AS
    EXEC('CREATE TABLE '+@userName+
        ' (column1 varchar(100), column2 varchar(100))');
GO
```

It is obvious that whatever `@userName` contains will be appended to the `CREATE` statement. An attacker could inject into the application some code that sets the user name to be:

```
a(c1 int); SHUTDOWN WITH NOWAIT;--
```

which will immediately stop the SQL Server without waiting for other requests to complete.

It is important to validate the input to ensure that no illegal characters are present. The application could be set to ensure that spaces are not permitted as part of the user name and this could be rejected before it ever got as far as constructing the `CREATE` statement.

If the stored procedure is going to construct a SQL command based on an existing object, such as a table or view, then it should check that such an object exists. For instance:

```
CREATE PROCEDURE dbo.AlterUserTable
    @userName sysname
AS
    IF EXISTS(SELECT * FROM INFORMATION_SCHEMA.TABLES
```

```
        WHERE TABLE_SCHEMA = 'dbo'
        AND TABLE_TYPE = 'BASE TABLE'
        AND TABLE_NAME = @userName)
    BEGIN
        // The table is known to exist
        // construct the appropriate command here
    END
GO
```

Error Messages

Error messages are useful to an attacker because they give additional information about the database that might not otherwise be available. It is often thought of as being helpful for the application to return an error message to the user if something goes wrong so that if the problem persists they have some useful information to tell the technical support team. Applications will often have some code that looks like this:

```
try
{
    // Attempt some database operation
}
catch(Exception e)
{
    errorLabel.Text = string.Concat("Sorry, your request failed. ",
        "If the problem persists please report the following message ",
        "to technical support", Environment.Newline, e.Message);
}
```

A better solution that does not compromise security would be to display a generic error message that simply states an error has occurred with a unique ID. The unique ID means nothing to the user, but it will be logged along with the actual error diagnostics on the server which the technical support team has access to. The code above would change to something like this instead:

```
try
{
    // Attempt some database operation
}
catch(Exception e)
{
    int id = ErrorLogger.LogException(e);
    errorLabel.Text = string.Format("Sorry, your request Failed. "+
        "If the problem persists please report error code {0} "
        "to the technical support team.", id);
}
```

Summary

- Encrypt sensitive data.
- Access the database using an account with the least privileges necessary.
- Install the database using an account with the least privileges necessary.
- Ensure that data is valid.
- Do a code review to check for the possibility of second-order attacks.
- Use parameterised queries.
- Use stored procedures.

- Re-validate data in stored procedures.
- Ensure that error messages give nothing away about the internal architecture of the application or the database.

References

- SQL Injection Attacks by Example shows the thought process of someone attempting a security review for the first time on a customer's intranet site.
- How To Use Forms Authentication with SQL Server 2000 demonstrates creating an authentication mechanism that stores passwords as a salted hash value.