CHAPTER

SQL Injection Attacks

9

INFORMATION IN THIS CHAPTER:

- What is a SQL injection attack?
- Why are SQL injection attacks so successful?
- How to figure out you have been attacked?
- How to protect yourself from a SQL injection attack?
- Cleaning up the database after a SQL injection attack

WHAT IS AN SQL INJECTION ATTACK?

An SQL Injection Attack is probably the easiest attack to prevent, while being one of the least protected against forms of attack. The core of the attack is that an SQL command is appended to the backend of a form field in the web or application front end (usually through a website), with the intent of breaking the original SQL Script and then running the SQL script that was injected into the form field. This SQL Injection most often happens when you have dynamically generated SQL within your front-end application. These attacks are most common with legacy Active Server Pages (ASP) and Hypertext Preprocessor (PHP) applications, but they are still a problem with ASP.NET web-based applications. The core reason behind an SQL Injection attack comes down to poor coding practices both within the front-end application and within the database stored procedures. Many developers have learned better development practices since ASP.NET was released, but SQL Injection is still a big problem between the number of legacy applications out there and newer applications built by developers who did not take SQL Injection seriously while building the application.

As an example, assume that the front-end web application creates a dynamic SQL Script that ends up executing an SQL Script similar to that shown in Example 9.1.

EXAMPLE 9.1

A simple dynamic SQL statement as expected from the application.

```
SELECT * FROM Orders WHERE OrderId=25
```

This SQL Script is created when the customer goes to the sales order history portion of the company's website. The value passed in as the OrderId is taken from the query string in the URL, so the query shown above is created when the customer goes to the URL http://www.yourcompany.com/orders/orderhistory.aspx?Id=25. Within the .NET code, a simple string concatenation is done to put together the SQL Query. So any value that is put at the end of the query string is passed to the database at the end of the select statement. If the attacker were to change the query string to something like "/orderhistory.aspx?id=25; delete from Orders," then the query sent to the SQL Server will be a little more dangerous to run as shown in Example 9.2.

EXAMPLE 9.2

A dynamic SQL String that has had a delete statement concatenated to the end of it.

```
SELECT * FROM Orders WHERE ORderId=25; delete from Orders;
```

The way the query in Example 9.2 works is that the SQL database is told via the semicolon ";" that the statement has ended and that there is another statement that should be run. The SQL Server then processes the next statement as instructed.

While the initial query is run as normal now, and without any error being generated but when you look at the Orders table, you would not see any records in the Orders table because the second query in that batch will have executed against the database as well. Even if the attacker omits the value that the query is expecting, they can pass in "; delete from Orders;" and while the first query attempting to return the data from the Orders table will fail, the batch will continue moving on to the next statement, which will delete all the records in the Orders table.

Many people will inspect the text of the parameters looking for various keywords in order to prevent these SQL Injection attacks. However, this only provides the most rudimentary protection as there are many, many ways to force these attacks to work. Some of these techniques include passing in binary data, having the SQL Server convert the binary data back to a text string, and then executing the string. This can be proven by running the T-SQL statement shown in Example 9.3.

EXAMPLE 9.3

Code showing how a binary value can be used to hide a T-SQL statement.

```
DECLARE @v varchar(255)
SELECT @v = cast(0x73705F68656C706462 as varchar(255))
EXEC (@v)
```

NOTE

The Database Is Not the Only Weak Spot

If a file name is going to be generated based on the user's input, a few special values should be watched for. These values are Windows file system keywords that could be used to give attackers access to something they should not have, or could simply cause havoc on the front-end server.

- AUX
- CLOCK$
- COM1-COM8
- CON
- CONFIG$
- LPT1-LPT8
- NUL
- PRN

By allowing an attacker to create a file path using these special names, attackers could send data to a serial port by using COM1 (or whatever com port number they specify) or to a printer port using LPT1 (or whatever printer port they specify). Bogus data could be sent to the system clock by using the CLOCK$ value, or they could instruct the file to be written to NUL, causing the file to simply disappear.

When data is being accepted from a user, either a customer or an employee, one good way to ensure that the value would not be used for an SQL Injection attack is to validate that the data being returned is of the expected data type. If a number is expected, the front-end application should ensure that there is in fact a number within the value. If a text string is expected, then ensure that the text string is of the correct length, and it does not contain any binary data within it. The front-end application should be able to validate all data being passed in from the user, either by informing the user of the problem and allowing the user to correct the issue, or by crashing gracefully in such a way that an error is returned and no commands are sent to the database or the file system. Just because users should be sending up valid data does not mean that they are going to. If users could be trusted, most of this book would not be needed.

The same technique shown in Example 9.3 can be used to send update statements into the database, causing values to be places within the database that will cause undesirable side effects on the websites powered by the databases. This includes returning javascript to the client computers causing popups that show ads for other projects, using HTML iframes to cause malicious software to be downloaded, using HTML tags to redirect the browser session to another website, and so on.

SQL Injection attacks are not successful against only applications which were built in-house, a number of third-party applications available for purchase are susceptible to these SQL Injection attacks. When purchasing third-party applications, it is often assumed that the product is a secure application that is not susceptible to the attack. Unfortunately, that is not the case, and any time a third-party application is brought into a company, it should be reviewed, with a full code review if possible, to ensure that the application is safe to deploy. When a company deploys a third-party

application that is susceptible to attack and that application is successfully attacked, it is the company that deployed the application that will have to deal with the backlash for having an insecure application and their customer data compromised, not the company that produced and sold the insecure application.

Many people think that SQL Injection attacks are a problem unique to Microsoft SQL Server, and those people would be wrong. SQL Injection attacks can occur against Oracle, MySQL, DB2, Access, and so on. Any database that allows multiple statements to be run in the same connection is susceptible to an SQL Injection attack. Now some of the other database platforms have the ability to turn off this function, some by default and some via an optional setting. There are a number of tickets open in the Microsoft bug-tracking website http://connect.microsoft.com that are requesting that this ability be removed from a future version of the Microsoft SQL Server product. While doing so would make the Microsoft SQL Server product more secure, it would break a large number of applications, many of which are probably the ones that are susceptible to SQL Injection attacks.

Another technique that is easier to use against Microsoft SQL Server 7 and 2000 is to use the sp_makewebtask system stored procedure in the master database. If the attacker can figure out the name of the web server, which can usually be done pretty easily by looking at the sysprocesses table, or the path to the website, then the sp_makewebtask procedure can be used to export lists of objects to HTML files on the web server to make it easier for the attacker to see what objects are in the database. Then they can simply browse to the website and see every table in the database.

EXAMPLE 9.4

Code that an attacker could execute to export all table objects to an HTML file.

```
exec master.dbo.sp_makewebtask '\\web1\wwwroot\tables.html",
"select * from information_schema.tables"
```

If xp_cmdshell is enabled on the server, then an attacker could use xp_cmdshell to do the same basic thing just by using Bulk Copy Protocol (BCP) instead of sp_makewebtask. The advantage to sp_makewebtask is that xp_cmdshell does not need to be enabled, while the downside to sp_makewebtask is that it does not exist on Microsoft SQL Server 2005 and up. The downside to xp_cmdshell is that, unless the application uses a login that is a member of the sysadmin fixed server role, the xp_cmdshell procedure will only have the rights that are granted by the proxy account. An attacker can use the xp_cmdshell procedure to send in the correct commands to give the account that is the proxy account more permissions, or even change the account to one that has the correct permissions. At this point BCP can be used to output whatever data is wanted. The attacker could start with database schema information, and then begin exporting your customer information, or they could use this information to change or delete the data from the database.

NOTE

There Are Lots of Protection Layers to Make Something Secure

Hopefully, by now you are starting to see how the various layers of the Microsoft SQL Server need to be secured to make for a truly secure SQL Server. In this case we look specifically at how the NTFS permissions, xp_cmdshell proxy account, the Windows account that the SQL Server is running under, the application account that logs into SQL having the minimum level of rights, and properly parameterizing the values from the web application all to create a more secure environment.

To fully protect from an SQL Injection attack, the application account should only have the minimum rights needed to function; it should have no rights to xp_cmdshell. In fact xp_cmdshell which should be disabled (or removed from the server). The SQL Server service should be running under a domain or local computer account that only has the rights needed to run as a service and access the SQL Server folders. That Windows account should have no rights to the actual files that are the website files, and it should not be an administrator on the server that is running the SQL Server service or the web server. The resulting effective permissions that a SQL Server has are to access the database files and do nothing else. Any other functions that the SQL Server instance is expected to perform either via an SQL Agent job or a CLR procedure should be controlled through some sort of account impersonation.

At the application level the actual SQL Server error messages should be masked so that they are not returned to the client. If you have done these things, then even if attackers were able to successfully complete an SQL Injection attack against the SQL Server they would not be able to do much to the server as they would not have any way to get information back from the SQL Server (as the error messages are masked) and they would not be able to get to the shell and get software downloaded and installed from the outside. Once these things fail a few times, attackers will probably just move on to an easier target.

The amount of time that an attacker will spend trying to successfully use an SQL Injection attack against a web-based application will for the most part depend on the amount of value the target has. A smaller company such as a wholesale food distributor probably would not be attacked very often, and the attacker will probably leave after a short period of time. However, a bank or other financial company will provide a much more enticing target for the attacker, and the attack will probably last much longer, with many more techniques being tried, as well as many combinations of techniques until they successfully break into the database application.

The catch to either of these techniques is that the NT File System (NTFS) permissions need to allow either the SQL Server account or the account that the xp_cmdshell proxy account uses to have network share and NTFS permissions to the web server. On smaller applications where the web server and the database server are running on the same machine, this is much, much easier as the SQL Server is probably running as the local system account that gives it rights to everything.

WHY ARE SQL INJECTION ATTACKS SO SUCCESSFUL?

SQL Injection attacks are so successful for a few reasons, the most common of which is that many newer developers simply do not know about the problem. With project timelines being so short, these junior developers do not have the time to research the security implications of using dynamic SQL. These applications then get left in

NOTE

Leaving Notes for Future Generations

Would not it have been great if when you were first learning about writing application code that talked to the database, if someone had told you how to properly write parameterized code? Even if that did not happen, it can happen for the next guy. Leave some comments in the application source code that you maintain, that explain why you are parameterizing the database code the way that you are so that the next guy can learn from how you have done things.

production for months or years, with little to no maintenance. These developers can then move through their career without anyone giving them the guidance needed to prevent these problems.

Now developers are not solely to blame for SQL Injection attack problems. The IT Management should have policies in place in order to ensure that newer developers that come in do not have the ability to write dynamic inline SQL against the database engine; and these policies should be in forced by code reviews to ensure that things are being done correctly. These policies should include rules like the following.

1. All database interaction must be abstracted through stored procedures.
2. No stored procedure should have dynamic SQL unless there is no other option.
3. Applications should have no access to table or view objects unless required by dynamic SQL, which is allowed under rule #2.

WARNING

SQL Injection Happens at All Levels

Unfortunately, not just small companies can have problems with SQL Injection attacks. In 2009, for example, ZD Net reported that some of the international websites selling Kaspersky antivirus, specifically Kaspersky Iran, Taiwan, and South Korea, were all susceptible to SQL Injection attacks. In the same article (http://bit.ly/AntiVirusSQLInject) ZD Net also reported that websites of F-Secure, Symantec, BitDeffender, and Kaspersky USA all had problems with SQL Injection attacks on their websites.

These are some of the major security companies of the day, and they are showing a total lack of security by letting their websites fall prey to the simple injection attack. Considering just how simple it is to protect a website from an SQL Injection attack, the fact that some of the biggest security companies in the industry were able to have SQL Injection problems on their production customer facing websites is just ridiculous.

Because of how intertwined various websites are with each other, real-estate listing providers and the realtors which get their data from the listing providers, a lot of trust must exist between these companies and the people who use one companies site without knowing that they are using another companies data. This places the company that is showing the real-estate listings to their users in a position of trusting the advertising company to have a safe application. However, this trust can backfire as on a few occasions various partner companies have suffered from SQL Injection attacks, in some cases pushing out malicious software to the users of dozens, hundreds, or thousands of different websites that display the data.

4. All database calls should be parameterized instead of being inline dynamic SQL.
5. No user input should be trusted and thought of as safe; all user interactions are suspect.

With the introduction of Object Relational Mappings (ORM) such as Link to SQL and nHybernate, the SQL Injection problems are greatly lessened as properly done ORM code will automatically parameterize the SQL queries. However, if the ORM calls stored procedures, and those stored procedures have dynamic SQL within them, the application is still susceptible to SQL Injection attacks.

HOW TO FIGURE OUT YOU HAVE BEEN ATTACKED

There are two basics ways that SQL Injection attacks are used. The first is to query data from a database and the second is to change data within the database. When the attacker is performing a query only SQL Injection attack, detecting the SQL Injection attack is not the most straightforward thing to do. When an attacker does a good job executing a query only SQL Injection attack against your website there should be little to no evidence of a SQL Injection attack on the SQL Server database instance itself. The only evidence of the SQL Injection attack will be within the web server's logs, assuming that the web server is configured to log all requests to the website.

If logging is enabled then the web server will include within the logs any HTTP requests which are passed to the web servers. These requests will include the SELECT statements which were attempted to be executed against the database engine. Analyzing these logs should be done using some sort of automated process. Packages could be created in SQL Server Integration Services which could process the transaction logs for a smaller scale website which could be used to parse the log files looking for successful requests which include SQL Statements in the query string or POST requests. For larger websites it may be more efficient to process the logs using a NoSQL platform such as Hadoop or Microsoft's HD Insight version of Hadoop so that the workload can be processed by dozens or hundreds of nodes depending on the amount of web server log data which needs to be processed daily. Making the processing of the web servers logs harder is the fact that you need to look for more than just the SELECT keyword. Remember from Example 9.3 where binary values were passed in from the attacker to the web server which were then converted by the SQL Server back to dynamic SQL? This attack vector also needs to be watched for by looking for keywords such as EXEC and CONVERT or CAST. Due to the rapid complexity of searching through the web server's logs scaling out this processing on a distributed system quickly becomes a much more flexible and scalable option.

SQL Injection attacks which change data are easier to identify as they leave a trail behind in the changed data which resides on the SQL Server Instance. Typically if an attacker is making changes to data within the SQL Server database they are going to change large amounts of data at once. One way to initially detect this is to look for

transaction logs which are suddenly larger than expected. This can give you a clue that may be a SQL Injection attack has modified large amounts of data within the database platform.

Other options include auditing data which should not be changing to see if that data has changed. If it has an alert can be sent to the database administrator or security team so that a person can visually inspect the data to see if it has been modified by normal processes or if invalid data, such as an HTML iframe tag has been injected into the data within the SQL Server's tables.

The third way to detect a SQL Injection attack against a SQL Server database where data has been changed is to monitor the web servers logs much like for the read only SQL Injection attack. While different key words may need to be evaluated this will still give you a good idea if something is attempting to attack the SQL Server instance.

There are unfortunately no standard SQL Server Integration Services packages of Hadoop queries which can be documented as each environment is different. These differences include the web server software which is used, the data points which are captured, the key words which need to be included or excluded based on the names of web pages, parameters which are expected, etc.

HOW TO PROTECT YOURSELF FROM AN SQL INJECTION ATTACK

Once the command gets to the database to be run by the database engine, it is too late to protect yourself from the SQL Injection attack. The only way to truly protect your database application from an Injection attack is to do so within the application layer. Any other protection simply would not be anywhere nearly as effective. Some people think that doing a character replacement within the T-SQL code will effectively protect you, and it might to some extent. But depending on how the T-SQL is set up and how the dynamic SQL string is built, it probably would not, at least not for long.

NET PROTECTION AGAINST SQL INJECTION

The only surefire way to protect yourself is to parameterize every query that you send to the database. This includes your stored procedure calls, as well as your inline dynamic SQL calls. In addition, you never want to pass string values that the front-end application has allowed the user to enter directly into dynamic SQL within your stored procedure calls. If you have cause to use dynamic SQL within your stored procedures (and yes, there are perfectly legitimate reasons for using dynamic SQL), then the dynamic SQL needs to be parameterized just like the code that is calling the stored procedure or inline dynamic SQL Script. This is done by declaring parameters within the T-SQL statement, and adding those parameters to the SQLCommand object that has the SQL Command that you will be running, as shown in Examples 9.5 and 9.6.

EXAMPLE 9.5

VB.NET code showing how to use parameters to safely call a stored procedure.

```
Private Sub MySub()
Dim Connection As SqlConnection
Dim Results As DataSet
Dim SQLda As SqlDataAdapter
Dim SQLcmd As SqlCommand
SQLcmd = New SqlCommand
SQLcmd.CommandText = "sp_help_job"
SQLcmd.CommandType = CommandType.StoredProcedure
SQLcmd.Parameters.Add("job_name", SqlDbType.VarChar, 50)
SQLcmd.Parameters.Item("job_name").Value = "test"
Connection = New SqlConnection("Data Source=localhost;Initial
Catalog=msdb;Integrated Security=SSPI;")
Using Connection
Connection.Open()
SQLcmd.Connection = Connection
SQLda = New SqlDataAdapter(SQLcmd)
Results = New DataSet()
SQLda.Fill(Results)
End Using
'Do something with the results from the Results variable here.
SQLcmd.Dispose()
SQLda.Dispose()
Results.Dispose()
Connection.Close()
Connection.Dispose()
End Sub
```

EXAMPLE 9.6

C# code showing how to use parameters to safely call a stored procedure.

```
private void MySub()
{
SqlConnection Connection = new SqlConnection("Data
Source=localhost;Initial Catalog=msdb;Integrated Security=SSPI;");
DataSet Results = new DataSet();
SqlCommand SQLcmd = new SqlCommand ();
SQLcmd.CommandText = "sp_help_job";
SQLcmd.CommandType = CommandType.StoredProcedure ;
SqlParameter parm1 = new SqlParameter();
parm1.ParameterName = "job_name";
parm1.DbType = DbType.String;
parm1.Precision = 255;
parm1.Value = "test";
SQLcmd.Parameters.Add(parm1);
Connection.Open();
SQLcmd.Connection = Connection;
SqlDataAdapter SQLda = new SqlDataAdapter(SQLcmd);
SQLda.Fill(Results);
//Do something with the results from the Results variable here.
SQLcmd.Dispose();
SQLda.Dispose();
Results.Dispose();
Connection.Close();
Connection.Dispose();
}
```

As you can see in the above, .NET code using a parameter to pass in the value is easy to do, adding just a couple of extra lines of code. The same can be done with an inline dynamic SQL string, as shown in Examples 9.7 and 9.8.

EXAMPLE 9.7

VB.NET code showing how to use parameters to safely call an inline dynamic SQL String.

```
Private Sub MySub()
Dim Connection As SqlConnection
Dim Results As DataSet
Dim SQLda As SqlDataAdapter
Dim SQLcmd As SqlCommand
SQLcmd = New SqlCommand
SQLcmd.CommandText = "SELECT * FROM dbo.sysjobs WHERE name=
@job_name"
SQLcmd.Parameters.Add("job_name", SqlDbType.VarChar, 50)
SQLcmd.Parameters.Item("job_name").Value = "test"
SQLcmd.CommandType = CommandType.Text;
Connection = New SqlConnection("Data Source=localhost;Initial
Catalog=msdb;Integrated Security=SSPI;")
Using Connection
Connection.Open()
SQLcmd.Connection = Connection
SQLda = New SqlDataAdapter(SQLcmd)
Results = New DataSet()
SQLda.Fill(Results)
End Using
'Do something with the results from the Results variable here.
SQLcmd.Dispose()
SQLda.Dispose()
Results.Dispose()
Connection.Close()
Connection.Dispose()
End Sub
```

EXAMPLE 9.8

C# code showing how to use parameters to safely call an inline dynamic SQL String.

```
private void MySub()
{
SqlConnection Connection = new SqlConnection("DataSource=
localhost;Initial Catalog=msdb;Integrated Security=SSPI;");
DataSet Results = new DataSet();
SqlCommand SQLcmd = new SqlCommand ();
SQLcmd.CommandText = "SELECT * FROM dbo.sysjobs WHERE name =
```

```
@job_name";
SQLcmd.CommandType = CommandType.Text;
SqlParameter parm1 = new SqlParameter();
parm1.ParameterName = "job_name";
parm1.DbType = DbType.String;
parm1.Precision = 255;
parm1.Value = "test";
SQLcmd.Parameters.Add(parm1);
Connection.Open();
SQLcmd.Connection = Connection;
SqlDataAdapter SQLda = new SqlDataAdapter(SQLcmd);
SQLda.Fill(Results);
//Do something with the results from the Results variable here.
SQLcmd.Dispose();
SQLda.Dispose();
Results.Dispose();
Connection.Close();
Connection.Dispose();
}
```

Once each parameter that is being passed into the database has been protected, the .NET code (or whatever language is being used to call the database) becomes safe. Any value that is passed from the client side to the database will be passed into the database as a value to the parameter. In the example code shown at the beginning of this chapter, the string value that has been passed into the application would then force an error to be returned from the client Microsoft SQL Server database as the value would be passed into a parameter with a numeric data type.

Using the sample query shown in the .NET sample code in Examples 9.7 and 9.8, if the user were to pass in similar attack code to what is shown in the SQL Server

NOTE

Do Not Trust Anything or Anyone!

The golden rule when dealing with SQL Injection is to not trust any input from the website or front-end application. This includes hidden fields and values from dropdown menus. Nothing should be passed from the front end to the database without being cleaned and properly formatted, as any value passed in from the front end could be compromised.

Hidden fields are probably the SQL Injection attacker's best friend. Because they are hidden from the end user's view and are only used by system processes, they are sometimes assumed to be safe values. However, changing the values that are passed in from a safe value to a dangerous value is a trivial matter for a script kitty, much less a skilled attacker.

When dealing with SQL Injection, the mantra to remember is NEVER, ever, trust anything that is sent to the application tier from the end user, whether or not the end user knows that he submitted the value.

sample code in Example 9.2, the query that would be executed against the database would look like the one shown in Example 9.9. This resulting query is now safe to run as the result which is executed against the database engine contains all the attack code as a part of the value.

EXAMPLE 9.9

Sample T-SQL code showing the resulting T-SQL Code that would be executed against the database if an attacker were to put in an attack code against the prior sample .NET code.

```
SELECT * FROM dbo.sysjobs WHERE name = 'test; delete from Orders';
```

In the sample code you can see that while the attack code has been passed to the engine, it has been passed as part of the value of the WHERE clause. However, because this is within the value of the parameter, it is safe because the parameter is not executable. If attackers were to pass in the same command with a single quote in front of it, in an attempt to code the parameter, and then execute their own code, the single quote would be automatically doubled by the .NET layer when it passed to the SQL Server database again, leaving a safe parameter value as shown in Example 9.10.

EXAMPLE 9.10

The resulting T-SQL code that would be executed against the database when an attacker passes in an attack string with a single quote in an attempt to bypass the protection provided by the .NET application layer.

```
SELECT*FROM dbo.sysjobs WHERE name = 'test''; delete from Orders';
```

PROTECTING DYNAMIC SQL WITHIN STORED PROCEDURES FROM SQL INJECTION ATTACK

When you have dynamic SQL within your stored procedures, you need to use a double protection technique to prevent the attack. The same procedure needs to be used to protect the application layer and prevent the attack from succeeding at that layer. However, if you use simple string concatenation within your stored procedure, then you will open your database backup to attack. Looking at a sample stored procedure and the resulting T-SQL that will be executed against the database by the stored procedure; we can see that by using the simple string concatenation, the database is still susceptible to the SQL Injection attack, as shown in Example 9.11.

EXAMPLE 9.11

T-SQL stored procedure that accepts a parameter from the application layer and concatenates the passed-in value to the static portion of the string, executing whatever attack code the attacker wishes against the database engine.

```
CREATE PROCEDURE sel_OrdersByCustomer
@LastName VARCHAR(50)
AS
DECLARE @cmd NVARCHAR(8000)
SET @cmd = 'SELECT*
FROM Orders
JOIN Customers ON Orders.CustomerId = Customers.CustomerId
WHERE Customers.LastName = ''' + @LastName + ''''
EXEC (@cmd)
GO
/*The command that will be executed when the attacker passed in ';
DELETE FROM Orders.*/
SELECT*
FROM Orders
JOIN Customers ON Orders.CustomerId = Customers.CustomerId
WHERE Customers.LastName = 'Smith'; DELETE FROM Orders '
```

Because of the value attack, the value being passed into the SQL Server Engine is passed in through the application layer, and the SQL Server engine does as it is instructed to do, which is to run the query. However, if we parameterize the dynamic SQL within the stored procedure, then the execute SQL code will be rendered harmless just as it would be if the dynamic SQL was executed against the database by the application layer. This is done via the sp_executesql system stored procedure as shown in Example 9.12.

EXAMPLE 9.12

T-SQL stored procedure that accepts a parameter from the application layer and uses parameterization to safely execute the query passing whatever attack code the users input safely against the database as a simple string value.

```
CREATE PROCEDURE sel_OrdersByCustomer
@LastName VARCHAR(50)
AS
DECLARE @cmd NVARCHAR(8000)
SET @cmd = 'SELECT *
FROM Orders
JOIN Customers ON Orders.CustomerId = Customers.CustomerId
WHERE Customers.LastName = ''' + @LastName + ''''
EXEC sp_executesql @cmd, '@LastName VARCHAR(50)', @LastName=@LastName
GO
/*The command that will be executed when the attacker passed in ';
DELETE FROM Orders.*/
SELECT *
FROM Orders
JOIN Customers ON Orders.CustomerId = Customers.CustomerId
WHERE Customers.LastName = 'Smith''; DELETE FROM Orders '
```

USING "EXECUTE AS" TO PROTECT DYNAMIC SQL

One of the best security related features within the SQL Server database engine is the "EXECUTE AS" T-SQL Syntax. This feature allows you to impersonate another login (at the instance level) or another user (at the database level) giving the administrator the ability to test commands and object access by using another, usually lower privileged, account without needing to have the password for the account. In order to impersonate another account within the SQL Server instance the login or user which is going the impersonation must have the right to impersonate the other account within the SQL Server instance. The IMPERSONATION right can be granted at the login level or the user level to Windows users, Windows groups or SQL Server users.

The "EXECUTE AS" feature is actually made up of two different commands. The first being "EXECUTE AS" and the second being "REVERT." The "EXECUTE AS" command (shown in Examples 9.12 and 9.13 later in this section) is used to begin the impersonation of another set of credentials; while the "REVERT" command (also shown in Examples 9.13 and 9.14 later in this section) is used to revert from the impersonated credentials back to the prior credentials. The prior credentials could be either the original credentials which were used to log into the database engine or a different set of credentials which were impersonated with a prior "EXECUTE AS" statement. It is important to note that "REVERT" may not always bring you back to the original credentials which the instance was logged into as "EXECUTE AS" statements can be nested several layers deep and there is no counter which can be used to identify how many iterations of "EXECUTE AS" have been entered through.

When using the "EXECUTE AS" the account remains impersonated for the lifetime of the session within the context of the execution of the "EXECUTE AS" statement. This means that if you run "EXECUTE AS" within a SQL Server Management Studio window all commands within that window will be executed within the context of those credentials until the "REVERT" command is run. When the "EXECUTE AS" statement is used within a stored procedure to impersonate an account the impersonated account will remain for the duration of the stored procedures execution or until the "REVERT" statement within the stored procedure, whichever comes first.

Impersonating a Login

As mentioned in the previous section, the EXECUTE AS statement can be used to impersonate logins at the instance level. This allows you to have the rights of the impersonated account within that query window. To demonstrate this functionality we can create a new login which has no extra permissions, and then use "EXECUTE AS" to impersonate this login then query for the list of available logins on the instance by querying the sys.server_principals system catalog view as shown in Example 9.13.

EXAMPLE 9.13

Sample code showing the creation of a new login then the use of that login using the "EXECUTE AS" T-SQL sytax.

```
IF EXISTS (SELECT * FROM sys.server_principals WHERE name =
'anotherLogin')
       DROP LOGIN anotherLogin
GO
CREATE LOGIN anotherLogin WITH PASSWORD='password1'
GO
EXECUTE AS LOGIN='anotherLogin'
SELECT * FROM sys.server_principals
REVERT
GO
```

When running the sample code as shown in Example 9.13 a result set will be returned which shows the "sa" login, the "anotherLogin" login and the various server roles (more about fixed server roles can be found in Chapter 12) on the instance. If the same select statement is run by a member of the sysadmin fixed server role that then more (possibly many more) logins will be shown including all the certificate logins which are created by default, and the various "NT SERVICE" accounts which are created automatically when the various SQL Server services are run under the local system accounts (more about these local accounts can be found in Chapter 12).

Impersonating a User

In addition to impersonating logins at the instance level, users can also be impersonated at the database level as well. This allows you access to the objects which the user has within the database as well as rights to objects which can be accessed in other databases via the public role or the guest account. In the example shown in Example 9.14 we create a new database, the change into the new database; creating a new loginless user within the database using that user to query the sys.databases system catalog view. The resulting record set includes only the master, tempdb and the new Impersonation databases as the modem and msdb databases are not accessable via through the public role.

EXAMPLE 9.14

Using the "EXECUTE AS" statement within a database to impersonate a database user.

```
use master
GO
IF EXISTS (SELECT * FROM sys.databases WHERE name = 'Impersonation')
       DROP DATABASE Impersonation
GO
```

```
CREATE DATABASE Impersonation
GO
use Impersonation
GO
IF EXISTS (SELECT * FROM sys.database_principals WHERE name =
'anotherUser')
        DROP USER anotherUser
GO
CREATE USER anotherUser WITHOUT LOGIN
GO
EXECUTE AS USER='anotherUser'
SELECT * FROM sys.databases
REVERT
GO
```

REMOVING EXTENDED STORED PROCEDURES

In addition to running all code from the application layer as parameterized commands instead of dynamically generated T-SQL, you should also remove the system procedures that can be used to export data. The procedures in question that you will want to remove are xp_cmdshell, xp_startmail, xp_sendmail, sp_makewebtask, and sp_send_dbmail. You may also want to remove the procedures that configure Database Mail such as sysmail_add_account_sp and sysmail_add_profileaccount_sp, so that attackers cannot use these procedures to give themselves a way to email out information from the database. Of course, you will want to make sure that you are not using these procedures in any released code and that you have Database Mail configured before removing your ability to configure it.

Of course, removing system stored procedures poses a risk of causing system upgrades to fail, so you will want to keep copies of these objects handy so that you can put the objects back before database version upgrades.

Unfortunately, this is not a surefire way to prevent an attacker from using these procedures. Crafty attackers can actually put these procedures back after they see that they have been removed. This is especially true of the extended stored procedures called DLLs (Dynamic Link Libraries), which must be left in their normal locations because other extended stored procedures that you do not want to remove are part of the same DLLs. The only saving grace is that you have to be a highly privileged user within the database engine to put an extended stored procedure into the SQL Server engine. Thus, the only way that an attacker could successfully put the extended stored procedures back would be to log into the database with a highly privileged account. If your application logs into the database engine using a highly privileged account, all bets are off as the attacker now has the rights needed to put the extended stored procedures back.

NOT USING BEST PRACTICE CODE LOGIC CAN HURT YOU

The application login process is probably the most important one that an attacker may want to take advantage of. Many times when developers are building a login

process within their application, the front-end developer will use a query similar to the one shown in Example 9.15. After the query shown in Example 9.15 is run, if there is a record in the record set then the user logged in correctly so we grab the values from the first row and move on.

EXAMPLE 9.15

Sample query for authentication

```
SELECT * from dbo.Users WHERE username = 'Something'
and password = 'something'
```

Attackers wishing to exploit this situation would be able to get past the login screen, probably being logged in with a high level of permissions. This is done by adding a small text string in the username field such as "user OR 1=1 –"What this will do is change the code shown in Example 9.16 into the code shown in Example 9.17. Example 9.18 shows the T-SQL code that would be executed against the database engine.

EXAMPLE 9.16

The way a sample record set looks when validating a user account.

```
SELECT * FROM dbo.Users WHERE UserName = 'user' AND
Password = 'password'
```

EXAMPLE 9.17

The way the code looks when the attack code has been inserted.

```
SELECT * FROM dbo.Users WHERE UserName = 'user' OR 1=1 -- AND
Password = 'password'
```

EXAMPLE 9.18

The executable part of the code against the database engine from the prior sample code.

```
SELECT * FROM dbo.Users WHERE UserName = 'user' OR 1=1
```

Because of the OR clause in the prior sample code, it does not matter if there is a record where the UserName column equals user because the 1 = 1 section will tell the database to return every record in the database.

As you can see in the sample code above, the code that gets executed against the database engine would return the entire User table. Assuming that the front-end application simply takes the first record from the record set returned from the database, the attacker would then be logged into the application, probably with an administrative-level account. Preventing this sort of attack is easy; refer back to the beginning of this section of this chapter for the sample .NET code. Now that the user has been logged in, potentially with administrative rights, the user does not need to use any additional dynamic SQL to get access to your customer data, as he or she will now have full access through your normal administrative system.

To combat this problem the application should be verifying that the values within the record set which are returned are actually the values which are expected by comparing the value in the username column to ensure that it matches the username which the user entered. If the two values then match the user is allowed to continue. If the usernames do not match, then we know that something has gone horribly wrong, a generic error should be returned to the end user, and alerts should be triggered within the application which alert the systems administration team that something has gone wrong.

WHAT TO RETURN TO THE END USER

The next important thing to configure within the front-end application is what errors are returned to the end user. When the database throws an error, you should be sure to mask the error from the end user. The end user does not have any need to know the name of either the primary key or the foreign key that has been violated. You might want to return something that the end user can give to customer service or the help desk so that the actual error message can be looked up.

What this has to do with SQL Injection is important. If the attacker is able to send in code that breaks the query and returns an error, the error may well contain the name of a table or other database object within the error message. For example, if the attacker sends in an attack string of "Group by CustomerId–" to a query that looks like "SELECT * FROM Customers WHERE UserName = 'UserName' AND Password = 'Password'," creating the query "SELECT * FROM Customers WHERE UserName = 'UserName' Group by CustomerId– AND Password = 'Password'." The default error message that SQL Server would return gives the attackers more information than they had before. It tells them the table name. The attacker can use this same technique to figure out which columns are in the table. Overall, being able to see the actual SQL Server error message, even if the error does not give the attacker any database schema information, it tells the attacker that the attack attempt was successful. By using the sp_MSforeachtable system stored procedure and the raiserror function, the attackers could easily return the list of every table in the database, giving them a wealth of information about the database schema, which could then be used in future attacks.

There is more useful information that an attacker could get thanks to the error message being returned. For example, if the users were to run a stored procedure in another database that they did not have access to, the error message would return the username of the user – for example, if the attacker sends in an attack string "; exec

NOTE

Why Are SQL Injection Attacks Still Possible?

One major reason why SQL Injection attacks are still possible today is that there is so much bad information circulating about how to protect yourself from an SQL Injection attack. For example, an article published by Symantec at http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks says that all you need to protect yourself is to verify the inputs using a regular expression that searches for the single quote and the double dash, as well as the strings "sp" and "xp." As you can see throughout this chapter, SQL Injection attacks can occur without tripping these regular expressions, and considering the high number of false positives that looking for a single quote would give you (especially if you like doing business with people of Irish descent), the protection would be minimal at best. If you were to read this article and follow its instructions you would be leaving yourself open to SQL Injection attacks.

model.dbo.Working –." It does not matter if the procedure exists or not, for the attacker would not get that far. The error returned from this call is shown in Example 9.19.

EXAMPLE 9.19

Error message returned by an attacker running a stored procedure that does not exist.

```
Msg 916, Level 14, State 1, Line 1
The server principal "test" is not able to access the database "model" under
the current security context.
```

The model database is an excellent database to try this against, as typically no users have access to the model database. If the attacker gets an error message saying that the procedure does not exist, the attacker now knows that the login that the application is logging into the database has some high-level permissions, or the model database has some screwed-up permissions.

After finding the username, the attacker can easily enough find the name of the local database that the application is running within. This can be done by trying to create a table in the database. This is because the error message when creating a table includes the database name. For example, if the attack code "; create table mytable (c1 int);--" is sent, the error message shown in Example 9.20 will be returned.

EXAMPLE 9.20

Error message returned when creating a table when you do no have rights returning the name of the database to the attacker.

```
Msg 262, Level 14, State 1, Line 1
CREATE TABLE permission denied in database "MyApplicationDatabase."
```

These various values can be used in later attacks to clear the database of its data or to export the data from the database.

DATABASE FIREWALLS

There are products available on the market today which are called database firewalls. These database firewalls are either software packages which run on the server which is running the SQL Server service, or they are an appliance which runs on its own hardware (or virtual machine). In either case the concept of these applications is simple, they intercept every SQL query which the application sends to the SQL Server's endpoint and if the application feels that the query is a SQL Injection query, based on the text of the query being sent in, the query will be terminated before the query ever gets to the SQL Server database engine.

While these appliances can do a good job with most basic SQL Injection attacks their ability to prevent more complex attacks is not well known. Because of this variable in the effectiveness of the database firewalls they should be part of the solution not the only solution for SQL Injection attacks.

TEST, TEST, TEST

One of the most important things that must be done when protecting a SQL Server database against a SQL Injection attack is to test the application regularly for SQL Injection vulnerabilities. This is can be done either internally by in house personal by using penetration testing applications or by outsourcing this function to trusted vendors who specialize in penetration testing. At the minimum penetration testing should be performed every time there is a major software package release, as well as quarterly to ensure that none of the minor releases which have been released during the quarter are susceptible to SQL Injection attacks.

In a perfect world penetration testing would be performed at every software release. But these penetration tests take time and resources, and the reality is that most companies do not have the resources available to do this sort of constant and consistent penetration testing.

CLEANING UP THE DATABASE AFTER A SQL INJECTION ATTACK

There are a few different attacks that an attacker can perform against an SQL Server database. As shown so far in this chapter, delete commands can be passed into the SQL engine. However, other commands can be executed as well. Usually, attackers do not want to delete data or take a system offline; they instead want to use the SQL Server to help launch other attacks. A simple method is to identify tables and

FAQ

IFRAME Versus PopUp

Often people ask if a popup blocker would prevent this iframe attack from affecting the end user, and the answer is no, it would not. An iframe does not show a web browser popup on the users screen. An iframe is an inline frame which shows within the displayed webpage. An iframe with a height of 0 would be totally invisible to the end user, but it could be requesting data from a webpage on another website, passing information from the user's computer back to this unknown website. The website that is called from the iframe could then exploit vulnerabilities in the end user's web browser to install key loggers or command and control software turning the end user's computer into a member of a bot-net.

NOTE

Notes About Using This Sample Code

Before running the included T-SQL code, be sure to make a full backup of the database in case of accidental data modification. The larger the database that you run this against, the longer it will take. When running this sample code, it is recommended that you change the output type from the default output style of grid to text by pressing <CTRL > + T, in order to reduce the resources needed to run the query. The included code will execute against all versions of Microsoft SQL Server from version 7 to 2014.

columns that are used to display data on the website that uses the database as a back-end. Then extra data is included in the columns of the database, which will allow attacking code to be executed against the database. This can be done using an update statement that puts an HTML iframe tag into each row of a table. This way when customers view the website, they get the iframe put into their web browser, which could be set to a height of 0 so that it is not visible. This hidden iframe could then install viruses or spyware on the user's computer without their knowledge.

Once this attack has occurred and viruses or spyware have been installed on the customer's computer, the most important thing now is to stop additional users' computers from being attacked. This means going through every record of every table looking for the attack code that is pushing the iframe to the customer's web browser. Obviously, you can go through each table manually looking for the records in question, or you can use the included sample code, shown in Example 9.21, which searches through each column in every table for the problem code and removes it. All you need to do is supply the variable with the attack code. The only columns that are not cleaned by this code are columns that use the TEXT or NTEXT data types. This is because the TEXT and NTEXT data types require special attention as they do not support the normal search functions.

NOTE

SQL Injection is Serious Business

In case you had not guessed after reading this chapter, SQL Injection attacks are a very serious threat. Normally when an SQL Injection attack is launched, it is not launched against a single website. An attacker will often write a program that will check as many websites as possible before the attacker is taken off the Internet. The last successful large-scale attack on the Internet (as of the writing of this book) successfully changed the data in tens of thousands of separate databases that run tens of thousands of different websites. At the time of the attack, this was verified by searching on Google for the text of the code that was inserted into the attacked databases and looking at the number of domains that Google returned with matches.

Falling prey to these attacks puts users and customers at major risk, as these attacks often are trying to install viruses or Trojan horses on the end user's computer, so that confidential data such as credit card numbers, and banking usernames and passwords can be gathered by the attacker and used to commit future fraud against the users and customers of the attacked websites. This can lead to months or years of credit report problems and the like.

One final point to keep in mind: If you use your own company's websites or services, then the SQL Injection attacker is attempting to attack you as you are also a customer. So do yourself a favor and protect the database so that you do not get viruses or Trojan horses installed on your computer through your own company's website.

EXAMPLE 9.21

T-SQL Code that will clean a database that has had its values updated to send unvalued code to users.

```
DECLARE @injected_value NVARCHAR(1000)
SET @injected_value = 'Put the code which has been injected here.'
/*Change nothing below this line.*/
SET @injected_value = REPLACE(@injected_value, '''', '''''')
CREATE TABLE #ms_ver (indexid INT, name sysname, internal_value INT,
character_value
VARCHAR(50))
INSERT INTO #ms_ver
EXEC xp_msver 'ProductVersion'
DECLARE @database_name sysname, @table_schema sysname, @table_name sysname,
@column_name
sysname, @cmd NVARCHAR(4000),
@internal_value INT
SELECT @internal_value = internal_value
FROM #ms_ver
DECLARE cur CURSOR FOR SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
COLUMN_NAME
FROM INFORMATION_SCHEMA.columns c
JOIN systypes st ON c.DATA_TYPE = st.name
WHERE xtype IN (97, 167, 175, 231, 239, 241)
OPEN cur
FETCH NEXT FROM cur INTO @database_name, @table_schema, @table_name,
@column_name
WHILE @@FETCH_STATUS = 0
BEGIN
SET @cmd = 'SELECT NULL
WHILE @@ROWCOUNT <> 0
BEGIN
'
IF @internal_value > 530000
SET @cmd = @cmd + ' SET ROWCOUNT 1000
UPDATE'
ELSE
SET @cmd = @cmd + ' UPDATE TOP (1000)'
SET @cmd = @cmd + ' [' + @database_name + '].[' + @table_schema + '].[' +
@table_name + ']
SET [' + @column_name + '] = REPLACE([' + @column_name + '], ''' +
@injected_value + ''', '''')
WHERE [' + @column_name + '] LIKE ''%' + @injected_value + '%''
END'
exec (@cmd)
FETCH NEXT FROM cur INTO @database_name, @table_schema,
@table_name, @column_name
END
CLOSE cur
DEALLOCATE cur
DROP TABLE #ms_ver
```

OTHER FRONT END SECURITY ISSUES

Any time the general public is invited to touch your servers (usually via a web browser, but it could be via any application which talks to a server) there is the potential for problems. This is because of the simple fact that people write the code, and people access the application. The first group of people are perfectly capable of making mistakes, or just being junior level professionals (even if they have senior level titles) and taking shortcuts in the code to get the project they are working on done. These shortcuts can lead to exploits which the second group of people can easily (though sometimes not so easily) exploit to get access to information that they should not have, or to damage in some way the information stored in the SQL Server (or other) database.

THE WEB BROWSER URL IS NOT THE PLACE FOR SENSITIVE DATA

One shortcut, which only the truly inexperienced developer will use is to put sensitive information into the URL bar of the client web browser. Some things, which should never be placed into the URL bar as part of the websites query string include:

- Username
- Password
- Account Number
- Social Security Number (or other national ID number)
- Driver's License Numbers
- Any kind of identifying information

The reason that you do not want to put this information into the URL bar is that would make it very easy for someone to change the value. This includes storing these values as hidden form fields being passed from one page to another. Proper secure application development practices tell us that we should only be storing this

WARNING

The SQL Server Session State Database Is Evil, Evil I Tell You

While there is no security issue with using the Microsoft Session State Database functionality which comes with Internet Information Server (IIS) and Microsoft SQL Server there is a major performance issue once the site starts to really grow. SQL Server was not designed to be a good key value store which is exactly what the session state database is. As the session state database begins to grow you will see all sorts of performance problems including ghost records, locking and blocking problems, index fragmentation issues (the indexes become basically 100% fragmented very quickly), IO problems, massive page splitting to name just a few. To get the best performance from Session State look at one of the in memory session state providers. The .NET framework includes one by default on all Windows Servers, it just needs to have its service started and there are several other options available. You may have noticed that I did not mention using the session state database in this chapter, and now you know why.

STORY TIME

It Is Not Just The Small Companies or the Junior Developers Who Make These Mistakes

In the middle of 2011, there was a little data disclosure issue at a small credit card and banking company called CitiBank. In the infinite wisdom of the developers at CitiBank they decided to include the account number of the person who was signed into the CitiBank portal in the URL bar, then use this handy piece of information to do lookups for the account information instead of relying of the username and password, or a session variable to hold this information.

Needless to say the results were pretty predictable. Someone figured out that once you signed into the CitiBank portal you could simply change the account number at the top and view someone else's account information. Based on all on and off the record answers it appears that the people that did this only gathered customer information, but the results could have been much, much worse. New credit cards could have been issued, accounts could have been closed, I general massive havoc could have been rained down upon the customers of CitiBank; all because a developer decided to stick a value in the URL bar and then trust that value.

information in a server side variable which is accessed via the automatically generated session information which the web server creates and manages.

The first excuse that people usually make at this point is that they need multiple web servers and the session information would not be available is the end user switches to another web server. The answer to this situation is actually very simple, its called using the .NET session state service (there are other third party session state services available as well).

If the website which is being developed continues to grow to MySpace (in its heyday, not today), Facebook, or Twitter sized then you need to find another solution than an in memory session state server. If you do not the in memory session state server will quickly become your bottleneck and another solution will need to be found as a single session state server (or even a session state farm) probably cannot effectively manage having hundreds of millions of records being accessed and modified regularly. In these cases it might make sense to either tie the user to the web server in question for the duration of the session visit, or store some small piece of information in an encrypted form, as well as the checksum or hash of that value in a cookie on the users system which can then be used to find the rest of the information that the user is looking for.

USING xEVENTS TO MONITOR FOR SQL INJECTION

Extended Events, also known as xEvents, are typically thought of as a performance monitoring tool not a security monitoring tool. However, there is not anything that says that Extended Events cannot be used to monitor for SQL Injection attacks.

While doing this in SQL Server 2008 or SQL Server 2008 R2 is not very easy, SQL Server 2012 introduced a bit of new xEvents functionality which makes this

NOTE

xEvents is Out of Scope of This Book

While diving into xEvents a little bit, this is not an xEvents book. For more information about xEvents look for any of the current books on xEvents such as THIS BOOK by Jonathan Kehayias.

possible. That functionality is the ability to define a filter against the text of the SQL Statement as shown in Example 9.22.

EXAMPLE 9.22

Creating an Extended Events Session to capture keywords in queries which may be issued against the SQL Server Database Instance in a SQL Injection Attack.

```
CREATE EVENT SESSION SQL_Injection_Trap
ON SERVER
ADD EVENT sqlserver.sql_statement_completed
        (ACTION (sqlserver.sql_text, sqlserver.plan_handle,
package0.collect_system_time)
        WHERE (sqlserver.sql_text LIKE '%SELECT%*%FROM%sys.tables%'
                        or sqlserver.sql_text LIKE '%DROP%'
                        or sqlserver.sql_text LIKE '%SQL_Injection_Trap%')
                )
ADD TARGET package0.asynchronous_file_target
   (SET FILENAME = N'c:\xEvents\SQL_Injection_Trap.xel', METADATAFILE =
N'c:\xEvents\SQL_Injection_Trap.xem')
WITH (EVENT_RETENTION_MODE = ALLOW_MULTIPLE_EVENT_LOSS)
GO
ALTER EVENT SESSION SQL_Injection_Trap
ON SERVER
STATE=start
GO
```

As the code in Example 9.22 is reviewed, special attention should be paid to the where clause within the ACTION portion of the declaration. This is where the clause that separates the functionality of SQL Server 2012 from the earlier releases of xEvents.

NOTE

Monitor for the xEvent Name

Whenever setting up anything which monitors for an attack like this, you should always monitor for the name of the monitor to see if anyone attempts to make changes to the monitor. In the case of the same code in Example 9.22 we monitor for the name of the event session so that if someone turns the session off we would capture the command which turns it off.

Viewing the data which is captured by this extended event is easy enough using the query shown in Example 9.23. A query like this could be scheduled to run hourly or nightly (changing the WHERE clause as needed) and if there are records returned send an alert to the database administration team or whoever is responsible for ownership of the SQL Server instance with the output of the query so that they can take action on the potential intrusion.

EXAMPLE 9.23

Showing sample code which reads the xEvents session capture.

```
SELECT *, data.value('(event/@timestamp)[1]', 'datetime'),
       data.value('(event/action[3])[1]', 'varchar(max)')
FROM
   (SELECT CONVERT (XML, event_data) AS data FROM sys.fn_xe_file_target_
read_file
      ('c:\xEvents\SQL_Injection_Trap*.xel', 'c:\xEvents\SQL_Injection_
Trap*.xem', NULL, NULL)
 ) entries
WHERE data.value('(event/action[3]/text)[1]', 'datetime')  BETWEEN
DATEADD(dd, -1, getutcdate()) AND GETUTCDATE()
order by data.value('(event/@timestamp)[1]', 'datetime') desc
GO
```

Sadly as of the writing of this book in Early 2014, Extended Events are not available in the Microsoft Azure SQL Database offering so this type of monitoring would not be available to users of the Microsoft Azure SQL Database platform.

SUMMARY

SQL Injection attacks pose some of the greatest dangers to the database and customers because they are typically used to directly affect the information that the customer sees and can be rather easily used to attempt to push malicious code to clients' computers. These attacks are very popular with attackers because they are a relatively easy way to exploit systems design. They are also popular because they are easy to reproduce once a site is found to be easy to compromise, as it usually takes a long time to correct all the potential attack points in a website. This length of time leaves the website and database open to attack for a long period of time as companies are usually unwilling to shut down their customer facing websites while the website design is being repaired.

Because of the way that the SQL Injection attacks work, the Database Administrator, Database Developer, Application Developer, and Systems Administrator all need to work together to ensure that they are correctly protecting the data within the

database and the company network at large. As the database and application developers begin getting in the habit of writing code that is not susceptible to SQL Injection attacks, the current project will become more secure, as will future projects that the team members work on.

SQL Azure is just as susceptible to an SQL Injection attack as any other SQL Instance. What the attacker can do within the instance is much less dangerous simply because there are many fewer features available. For example, protection against xp_cmdshell is not a priority because xp_cmdshell is not available on an SQL Azure instance. Neither are features such as database mail or SQL mail, so protecting against attackers that plan to use these vectors does not need to be done. As time goes on, and more features are added to Windows Azure SQL Database, this may change; however, as of this writing, this information is accurate.

With proper planning (such as proper object security design) and proper monitoring (such as using xEvents to capture suspect statements) we can prevent many of the potential SQL Injection problems and when those problems do show up, we can track the statements to see that something happened and what happened.

REFERENCE

Wall Street Journal, Others, Hit in Mass SQL attack – SC Magazine US. *IT Security News and Security Product Reviews – SC Magazine US*. n.d. Web. October 21, 2010.