## SQL Injection Attacks

This article is an extract from the book [Tribal SQL](http://rd.gt/1iR9Z4C). In this article, Kevin Feasel explains SQL injection attacks, how to defend against them, and how to keep your Chief Information Security Officer from appearing on the nightly news.

Imagine waking up one morning and, while surfing the Internet at breakfast, seeing news articles describing how a hacker stole your company's proprietary data and put it up on Pastebin. Millions of customer records, perhaps Protected Health Information, credit card numbers, or even names of sources, undercover agents, or employees, all out there for the world to see. This may be roughly the time when you crawl back into bed and start planning that long-dreamed-of trip to the Australian outback to start a new life as a nomad.

Before you reach the point where you are trying to explain to an interviewer the circumstances behind your sudden departure from your previous employment, perhaps I should divulge the single most pernicious résumé-updating experience (security flaw division) out there: SQL injection.

In September of 2011, Imperva's *Hacker Intelligence Initiative* released a report stating that SQL injection was responsible for 83% of successful data breaches from 2005 through to the report's release (<http://tinyurl.com/py8ltlf>). It should be no surprise that the Open Web Application Security Project (OWASP) rated injection attacks, and especially SQL injection, as the number one threat vector in 2010 (<http://tinyurl.com/pzkrlds>).

In every instance of SQL injection, the flaw is the same: an attacker injects SQL code in a manner the application's developers did not anticipate, allowing the attacker to perform unexpected and unauthorized actions. For a very simplistic example of this, imagine a basic web page with a search box on it. The web page developer expects users to search for a product and look up rows in a table based upon the contents of that search box. If the developer is not careful, the user may be able to craft a search string that returns all products – or something much worse, like returning a list of all of the tables in the database, or a list of patients' medical records in another database.

Now that we have a common understanding of the gravity of the problem, as well as a basic definition of SQL injection, the rest of this paper will go into further detail on how to perform SQL injection attacks, how to defend against them, and how to keep your Chief Information Security Officer from appearing on the nightly news.

**My First SQL Injection Attack**

If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.   
  
Sun Tzu, The Art of War (<http://gutenberg.org/cache/epub/132/pg132.html>)

You cannot truly defend against a threat you do not understand, so the first step in defending against a SQL injection attack is to understand precisely how one works. Before I begin, some standard provisos. Firstly, don't perform a SQL injection attack on any application without express, written permission. Second, don't put known unsafe code on a production machine. Finally, don't put your testing code in a publicly accessible location, as some bad person somewhere will probably find it eventually.

With that out of the way, the first example will be a very simple web page on top of the ubiquitous AdventureWorks database, specifically an ASP.NET web forms application talking to a SQL Server 2008 instance, but SQL injection is relevant across all application and database platform combinations.

Imagine a very basic grid showing a list of product subcategories from the Production.ProductSubcategory table in AdventureWorks. In addition to this list, we have a name filter in which a user can type a partial, with a grid that displays matching items. The SQL for such a query could look as shown in Listing 1.

DECLARE @Filter NVARCHAR(50);

SET @Filter = 'Bike';

SELECT ProductSubcategoryID ,

ProductCategoryID ,

Name

FROM Production.ProductSubcategory

WHERE Name LIKE '%' + @Filter + '%'

Listing 1: A filtered list of product subcategories.

The query returns rows of products that include "Bike" in their name. This is the expected behavior, and all is well. To simulate a SQL injection attack, we can try changing the filter value from 'Bike' to 'Bike'' OR 1=1--'. Our goal as attackers is to get "outside" the bounds of the parameter, at which point we can manipulate the query itself or run some completely different SQL statement. In this particular case, our goal is to extend the search surreptitiously, to return rows where the Name is like "%Bike" or all of the results (because 1 equals 1 is always true). We then comment out the rest of the query to prevent any syntax errors.

Running this first attempt at an attack will show no results, meaning our attack failed. The reason is that we were not able to get "outside" of the parameter, so instead of searching for "%Bike" or where 1=1, we are actually searching for a product subcategory whose name is like "Bike' OR 1=1--" and naturally, there are no product subcategories which match that name.

What we did was attempt to perform SQL injection against a static SQL query, which simply is not possible. SQL injection is only possible against dynamic SQL, either through an ad hoc statement put together by an application, which communicates with SQL Server, or through SQL Server's built-in dynamic SQL capabilities. Listing 2 constructs a basic dynamic SQL query that returns the same results as Listing 1, when used as intended.

DECLARE @Filter NVARCHAR(MAX);

SET @Filter = 'Bike';

DECLARE @sql NVARCHAR(MAX);

SET @sql = N'select ProductSubcategoryID, ProductCategoryID, Name

from Production.ProductSubcategory where Name like ''%'

+ @Filter + N'%'''

EXEC(@sql);

Listing 2: Returning a filtered list of product subcategories using dynamic SQL.

However, try the same attack (substituting in SET @Filter = 'Bike'' OR 1=1--';), and we see very different results. The query returns all of the subcategories, including entries such as Handlebars and Brakes. This is certainly not something that the procedure writer expected, and can have considerable negative ramifications. For example, if we change the filter as shown on Listing 3, we can see all of the table schemas and names.

DECLARE @Filter NVARCHAR(MAX);

SET @Filter = 'I do not care''

UNION ALL

select 1, 1, TABLE\_SCHEMA + ''.'' + TABLE\_NAME from INFORMATION\_SCHEMA.TABLES --';

;

DECLARE @sql NVARCHAR(MAX);

SET @sql = N'select ProductSubcategoryID, ProductCategoryID, Name

from Production.ProductSubcategory where Name like ''%'

+ @Filter + N'%'''

EXEC(@sql);

GO

Listing 3: Using SQL injection to view table schemas and names.

From there, we can perform reconnaissance on various tables and even do entirely unexpected things like inserting our own product subcategories or, even worse, dropping tables. In Listing 4, we take advantage of our reconnaissance work to insert a new product subcategory. We know that a product subcategory has four non-nullable, non-identity attributes: ProductCategoryID (of type integer), Name (a varchar), rowguid (a uniqueidentifier), and ModifiedDate (a datetime). The code in Listing 4 fills in all four columns with appropriate values, so that our malicious insert statement succeeds.

declare @Filter nvarchar(max);

set @Filter = 'i do not care'';

insert into Production.ProductSubcategory(ProductCategoryID, Name,

rowguid, ModifiedDate)

values(4, ''Evil Subcategory'', newid(), current\_timestamp);--';

Listing 4: Using SQL injection to add an "evil" subcategory.

If the account running the query has elevated privileges, it could possibly have access to other databases, allowing an attacker to collect information far more valuable than product subcategories.

**Attacking Websites**

In the previous section, we envisioned a basic site. In this section, we will build the site and attack it directly, applying what we learned in Management Studio. The full code for this project is available in .zip format, as part of the code download for this book.

Our website will be a rather simple ASP.NET web forms application. Listing 5 shows the base HTML.

<div>

<asp:TextBox ID="txtSearch" runat="server" Text="Enter some text here." />

<asp:Button ID="btnClickMe" runat="server" Text="Click Me" OnClick="btnClickMe\_Click" />

</div>

<br />

<div>

You searched for: <asp:Label ID="lblSearchString" runat="server" />

</div>

<br />

<div>

<asp:GridView ID="gvGrid" runat="server" AutoGenerateColumns="true">

<Columns>

<asp:TemplateField>

<HeaderTemplate>Name</HeaderTemplate>

<ItemTemplate><%# Eval("Name") %></ItemTemplate>

</asp:TemplateField>

</Columns>

</asp:GridView>

</div>

Listing 5: A simple ASP.NET web form.

Create a project in Visual Studio, and copy and paste this into a new page. Then, go to the code-behind. This code-behind is simplistic with a number of major errors, so please do not think of this as production-worthy code; neither should any of this be run in a production environment. With that said, Listing 6 provides sample C# code to query SQL Server based on an input search string.

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

//Using .NET 4.0 here. If you want to use 2.0/3.5, change to

//IsNullOrEmpty.

if (!String.IsNullOrWhiteSpace(Request.QueryString["search"]))

LoadData(Request.QueryString["search"]);

}

}

protected void btnClickMe\_Click(object sender, EventArgs e)

{

LoadData(txtSearch.Text);

}

protected void LoadData(string Filter)

{

//Using .NET 4.0 here. If you want to use 2.0/3.5, change to IsNullOrEmpty.

if (String.IsNullOrWhiteSpace(Filter))

{

Response.Write("You need a filter here, buddy!");

gvGrid.Visible = false;

}

else

{

using (SqlConnection conn = new

SqlConnection("server=localhost;database=AdventureWorks;

trusted\_connection=yes"))

{

string sql = String.Empty;

sql = "select Name, ProductSubcategoryID, ProductCategoryID from

Production.ProductSubcategory where Name like '%" + Filter +

"%' order by ProductSubcategoryID;";

using (SqlCommand cmd = new SqlCommand(sql, conn))

{

cmd.CommandTimeout = 30;

conn.Open();

SqlDataReader dr =

cmd.ExecuteReader(CommandBehavior.CloseConnection);

gvGrid.DataSource = dr;

gvGrid.DataBind();

gvGrid.Visible = true;

}

lblSearchString.Visible = gvGrid.Visible;

lblSearchString.Text = Filter;

}

}

}

Listing 6:  Bad C# code to search product subcategories by name.

Using this code, we now have a functional website we can use for SQL injection attacks. This website's flaws offer us two basic means for injection: through the querystring, or through a text-box and button. Both of these eventually call the LoadData function in the code-behind, which runs a SQL query to pull back a list of product subcategories, given a filter. Fire up a debug session in Visual Studio and start attacking the site through either mechanism.

Before performing an attack, it is typically a good idea to understand normal behavior. Type "bike" into the text box and click the button. You should see a grid with a list of five bike-related subcategories appear, as per Listing 1, previously. Unfortunately, with just a few more characters, we can begin to perform unexpected actions. Now search for the following:

'bike' OR 1=1-- '

After entering this code snippet into our text-box, we can see the entire list of product subcategories. From there, the world, or at least the database, is our oyster. We can use this page to perform reconnaissance on the SQL Server instance, searching for databases, tables, and even data. For example, the following query will show all of the databases on the instance, as well as their IDs and SQL Server compatibility level:

'nada' UNION ALL select name, database\_id, compatibility\_level

from sys.databases;--'

The latter two pieces of information are not necessarily important for an attack, but we needed something to ensure that the second half of our UNION ALL statement has a column structure that is compatible with the first half. This particular column structure includes one varchar and two integer fields, and so we follow along to match the schema and prevent an error from being returned.

Aside from running select statements that match the "expected" query's schema, we can perform other types of queries, simply by typing them into the available text box. For example, it would be possible to enable xp\_cmdshell, get the server to connect via FTP to another server and download malicious software, and then use xp\_cmdshell again to install this malicious software, thereby giving an attacker full control over a database server.

Alternatively, we could use xp\_cmdshell to open a web browser and navigate to a specific page that attempts to install malicious software on the server using JavaScript or an ActiveX component. Both of these attack mechanisms would have the same effect in terms of potentially compromising a database server. What follows is an example, showing the first step of this attack, turning on xp\_cmdshell (in case it is not already enabled).

'nada'; exec sp\_configure 'show advanced options', 1; reconfigure;

exec sp\_configure 'xp\_cmdshell', 1; reconfigure; --'

After turning on xp\_cmdshell, it would be possible to execute malicious code through xp\_cmdshell. This is not as easy as simply having it open an instance of Internet Explorer and navigating to one's malicious site of choice, but it is possible.

**Defending Websites**

Now that we have an idea of how to attack a website, we can start to formulate ideas for how to defend one. We'll consider three:

* Blacklists, and why they cause problems and don't work.
* Whitelists, and why they are essential in some cases, but don't work in others.
* Parameterized Queries, the best and most effective form of protection from SQL Injection.

**Blacklists**

The first line of defense that many developers come up with is a blacklist: we know that keywords like "select," "insert," and "drop" are necessary to perform a SQL injection attack, so if we just ban those keywords, everything should be fine, right? Alas, life is not so simple; this leads to a number of problems with blacklists in general, as well as in this particular case.

The second-biggest problem with blacklists is that they could block people from performing legitimate requests. For example, a user at a paint company's website may wish to search for "drop cloths," so a naïve blacklist, outlawing use of the word "drop" in a search would lead to false positives.

The biggest problem is that, unless extreme care is taken, the blacklist will still let through malicious code. One of the big failures with SQL injection blacklists is that there are a number of different white-space characters: hex 0x20 (space), 0x09 (tab), 0x0A, 0x0B, 0x0C, 0x0D, and 0xA0 are all legitimate white-space as far as a SQL Server query is concerned. If the blacklist is looking for "drop table," it is looking for the word "drop," followed by a 0x20 character, followed by the word "table." If we replace the 0x20 with a 0x09, it sails right by the blacklist.

Even if we do guard against these combinations, there is another avenue of attack.

**Why blacklists don't work**

Simply put, there are too many ways round them. We'll consider two of the main attack vectors that render blacklists impotent.

**HTTP Parameter Pollution**

[HTTP Parameter Pollution](http://tinyurl.com/HTTPpol) uses a quirk in HTML query strings to attack websites. A typical query string looks like the following:

**http://[website].com/SomePage.aspx?SearchTerm=Elvis&MemorabiliaType=Clock**

In this example, we are looking for an Elvis clock. The SearchTerm parameter for this example is set to Elvis, and the MemorabiliaType parameter is set to Clock. The web page can read these parameters (in ASP.NET, these are viewed with calls to Request.QueryString["SearchTerm"] and Request.QueryString["MemorabiliaType"], respectively) and handle them as necessary.

What happens, though, when somebody modifies the URL to add in a second SearchTerm parameter? The resulting behavior differs, depending upon which web platform is handling the URL request. The web platform could convert the different parameters into a list, ignore all but the first instance of a parameter, ignore all but the last instance of a parameter, display an error on the web page, or take any of a number of other actions. ASP.NET puts the different terms into a comma-delimited string. Thus, if we change the above URL to:

**http://[website].com/SomePage.aspx?SearchTerm=Elvis&SearchTerm=Evil&MemorabiliaType=Clock**

Request.QueryString["SearchTerm"] would return " Elvis,Evil" instead of just " Elvis" or just " Evil".

Armed with this knowledge, an attacker can use HTTP Parameter Pollution in a SQL injection attack to get around basic blacklist filtering. For example, suppose that the website does in fact filter out the word "drop" followed by any of the seven white-space characters, followed by the word "table." In that case, we could still perform a query-string-based attack and drop a table by putting /\* and \*/ around our HTTP query parameters to comment out the commas. This leaves us with a valid SQL statement:

SomePage.aspx?SearchTerm=**drop**/\*&SearchTerm=**\*/table dbo.Users;--**

There is no white space after the " drop" keyword, or before the " table" keyword, so this attack gets right around the blacklist rule described above; best of all, because HTTP Parameter Pollution is not well known among developers, they probably have not even thought of this particular behavior, leaving the website exposed while creating a false sense of security.

Queries as binary strings

Even if a developer has created a blacklist that plans around the HTTP Parameter Pollution attack vector, there are yet more methods available. For example, we can convert a SQL statement to binary, and then execute the binary. Suppose that we want to get a list of names, database IDs, and compatibility levels from all databases on the server, but we cannot use the " select" keyword at all because of a blacklist. One way around this limitation would be to get a binary representation of the query, which is easy to do in SQL Server Management Studio.

SELECT CAST('select name, database\_id, compatibility\_level

from sys.databases;' AS VARBINARY(8000));

Listing 7: Returning a binary representation of a query.

This returns a long hexadecimal string. Copy and paste that string in Listing 8 and you have an attack.

DECLARE @i VARCHAR(8000);

SET @i = CAST([long hex string] AS VARCHAR(8000));

EXEC(@i);

Listing 8: Using a binary string form of a query in an injection attack.

This attack, combined with HTTP Parameter Pollution, means that attackers can bypass more blacklists. In this particular case, the result would not be interesting, because this query would run separately from the SQL query that .NET runs, so the results would not be in the same data set and would not appear on our grid. With that said, however, there have been several attacks that used the execution of binary representations of SQL queries to update all textual columns on databases, inserting code into each row that tries to open malicious JavaScript on rogue servers.

**Whitelists**

Blacklists are an untenable option, so the next thought might be to switch to whitelists. A whitelist is the opposite of a blacklist: instead of expressly prohibiting specified elements, and allowing anything not on the list, a whitelist allows specified elements and prohibits everything else.

In some cases, whitelists are essential. For example, with a radio button list, the server side should check whether the form value returned is valid. If the only possible values are "0," "1," and "2," but the value "monkey" is received back, somebody has, well, monkeyed with the form; from there, the server could either use a default value or throw an exception. Also, if the structure of a text field is known, such as a Social Security number, credit card number, amount of money, or date, a whitelist can accept certain patterns, rejecting all others. This helps detect data entry errors and also protects against SQL injection in these particular form fields, at least until "'; drop table Users--" becomes a valid Social Security Number.

This kind of whitelist breaks down for general searches, though. With free-form text fields, there is no necessary pattern, and so we cannot create a regular expression against which to check the data. Thus, a whitelist is not a valid option in all circumstances.

**Parameterized queries**

Forget about blacklists, and move whitelisting into the world of data validation rather than SQL injection prevention. There is only one effective method for preventing SQL injection through an application: query parameterization.

In our sample code, we used a SqlCommand object to execute a SQL statement and return the results in a SqlDataReader. This worked but, as we have subsequently learned, it is not a safe way of doing things. The reason is that the Filter variable was not parameterized correctly. Instead, our data access code concatenated various strings, including the Filter parameter, to create one SQL query. The correct alternative to developer-led concatenation is to build a SqlParameter, put the contents of the Filter variable into the parameter, and pass that off to SQL Server as a contained parameter rather than simply being part of the text. This change is relatively simple, making our data access code look as shown in Listing 9.

using (SqlConnection conn = new SqlConnection("server=localhost;database=AdventureWorks;

trusted\_connection=yes"))

{

string sql = String.Empty;

sql = "select Name, ProductSubcategoryID, ProductCategoryID

from Production.ProductSubcategory

where Name like '%@Filter%' order by ProductSubcategoryID;";

using (SqlCommand cmd = new SqlCommand(sql, conn))

{

//create a parameter for @Filter

SqlParameter filter = new SqlParameter();

filter.ParameterName = "@Filter";

filter.Value = Filter;

//attach our parameter to the SqlCommand

cmd.Parameters.Add(filter);

cmd.CommandTimeout = 30;

conn.Open();

SqlDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);

gvGrid.DataSource = dr;

gvGrid.DataBind();

gvGrid.Visible = true;

}

//Continue along with our code

}

Listing 9: A **SqlParameter** object in C# eliminates our prior SQL injection vulnerability.

With this minor change in data access code, we eliminate the possibility of a SQL injection attack on this particular query. We can try any combination of characters or attack techniques but it will be to no avail: we are unable to get "outside" the parameter.

The way this works is that, when we have a SqlCommand object with associated SqlParameter objects, the query sent to SQL Server actually changes somewhat. Open up an instance of SQL Server Profiler and start a new trace using the T-SQL template, and then return to the sample web page. This web page simply creates a SQL string and does not parameterize the query. Using our classic "'bike' or 1=1--" SQL injection code, we can see the end result as follows:

select Name, ProductSubcategoryID, ProductCategoryID

from Production.ProductSubcategory

where Name like '%bike' or 1=1--%' order by ProductSubcategoryID;

This is just as we would expect; the query runs, and the user's unexpected SQL code changes its basic structure. In comparison, on a web page that parameterizes the query correctly, there is quite a different result.

exec sp\_executesql N'select Name, ProductSubcategoryID, ProductCategoryID

from Production.ProductSubcategory where Name like ''%@Filter%'' order by

ProductSubcategoryID;',N'@Filter nvarchar(14)',@Filter=N'bike'' or 1=1--'

We now see a call to sp\_executesql, passing in the Filter parameter we used earlier. We can also see that our apostrophe, the character used to escape the confines of the parameter, has been doubled up, thus making it safe. There is absolutely no way to perform a SQL injection attack here; each apostrophe is doubled up and the query will remain safe. Despite that, here are a few more words of wisdom for added protection and perhaps even better system performance.

Be sure to match field sizes to the size of the character data types, wherever possible. For example, suppose there is a column on a table defined as a varchar(10). In that case, the text box should only allow 10 characters of text. Even if the query were still susceptible to SQL injection, there are only a limited number of attacks possible with just 10 characters. You should also use matching field sizes in SqlParameter objects, setting a fixed size. As seen above, because we did not use a fixed size for the filter, the SqlParameter object's size was set to the length of the string: 14 characters. If somebody else types in 15 characters-worth of text, this creates a new execution plan and both end up in the plan cache. This is potentially a waste of precious plan cache space so, by having one specific size, SQL Server generates one plan in the cache, saving room for the plans of other queries.

**Protecting Stored Procedures**

At this point, I would like to leave the world of websites and return to the database server proper. We know that parameterized queries are the ultimate solution to SQL injection in our applications, web or otherwise. On the database side, however, we should look at stored procedures.

Fortunately, static stored procedures are, by themselves, invulnerable to SQL injection. For example, consider a stored procedure with the structure shown in Listing 10.

CREATE PROCEDURE GetProductSubcategoryByName

@ProductSubcategoryName NVARCHAR(MAX)

AS

SELECT ProductSubcategoryID ,

ProductCategoryID ,

Name ,

rowguid ,

ModifiedDate

FROM Production.ProductSubcategory

WHERE Name = @ProductSubcategoryName;

GO

Listing 10: A static GetProductSubcategoryByName stored procedure.

All of the attack attempts shown in Listing 11 will fail when run in Management Studio.

EXEC GetProductSubcategoryByName N''' OR 1 = 1--';

EXEC GetProductSubcategoryByName N''' UNION ALL select 1, 2, 3, 4, 5--';

EXEC GetProductSubcategoryByName N'''declare @i varchar(8000); set @i = CAST([binary truncated so you can read this more easily] as varchar(8000)); exec(@i);'

Listing 11: Some failed injection attacks on GetProductSubcategoryByName.

There is no way to "break out" of the parameter and so our query is safe from SQL injection. This is not a trivial result; it means that we can replace ad hoc SQL in our data layer with calls to stored procedures. This does not mean that ad hoc SQL is necessarily unsafe. The parameterized query in Listing 9 uses ad hoc SQL, and is immune from SQL injection. For this reason, I would not use the threat of SQL injection as a core reason for supporting stored procedures over ad hoc SQL in a .NET environment.

However, stored procedures do force you to use parameterization, whereas a developer might accidentally forget to parameterize an ad hoc SQL query. Unfortunately, there are ways to abuse stored procedures, such as using a .NET 1.1 SqlDataAdapter. This is an old and terrible method for getting data, and yet there is sample code out on the Internet that still uses it. Listing 12 shows an example of a dataset populated by a SqlDataAdapter.

//HINT: If your code looks like this, you're doing it wrong.

using (SqlConnection conn = new

SqlConnection("server=.;database=AdventureWorks;trusted\_connection=yes"))

{

DataSet ds = new DataSet();

SqlDataAdapter sda = new

SqlDataAdapter("Production.GetProductSubcategoryByName '" + Name + "'", conn);

//Problem #1: running a stored procedure as a regular SQL query.

**//sda.SelectCommand.CommandType = CommandType.StoredProcedure;**

//If you uncomment the line above, at least the query would fail.

sda.Fill(ds);

gvGrid.DataSource = ds;

gvGrid.DataBind();

}

Listing 12: Stored procedures can be vulnerable to SQL injection attacks.

As mentioned in the comments, by not setting the CommandType of the SqlDataAdapter to that of a stored procedure, the code is susceptible to SQL injection, even when only static stored procedures are used. If we uncomment the line of code that sets the command type, and try to run this same process, we would get an exception and the query would fail, rather than performing potentially malicious actions.

The more interesting case involves stored procedures that use dynamic SQL. When somebody hears that stored procedures are "safe" whereas ad hoc SQL is "unsafe," that person might go around creating stored procedures like the one in Listing 13.

CREATE PROCEDURE Production.VulnerableSearch @Filter VARCHAR(50)

AS

DECLARE @sql VARCHAR(250);

SET @sql = 'select \* from Production.ProductSubcategory

where Name like ''%'

+ @Filter + '%'';';

EXEC(@sql);

GO

Listing 13: A really bad stored procedure using dynamic SQL.

Code like that is just as susceptible to SQL injection as ad hoc SQL, because this stored procedure is essentially a vessel for executing ad hoc SQL. In fact, it is actually worse, because at least with legitimate ad hoc SQL, we can still parameterize the queries in .NET and turn them into good dynamic SQL calls, whereas with this procedure, we do not even get that benefit.

The stored procedure in Listing 13 is, admittedly, ridiculous, but more complicated stored procedures often make the same basic mistake. Take, for example, a stored procedure that retrieves a set based on some partial search criteria, or a procedure which takes as input a list of values rather than a single item, and returns all elements in that list. Listing 14 shows an example of the former.

CREATE PROCEDURE Production.VulnerableSearch @Filter VARCHAR(50)

AS

DECLARE @sql VARCHAR(250);

SET @sql = 'select \* from Production.ProductSubcategory

where Name like ''%'

+ @Filter + '%'';';

EXEC(@sql);

GO

Listing 14: Another bad stored procedure using dynamic SQL.

In general, attacking dynamic SQL is the same as attacking ad hoc SQL through a website: escape out of the current query, perform the attack, and comment out the rest of the expected query. We can perform a similar attack on this procedure through SQL Server Management Studio (or any other client that connects to the database), except this time, we have to double up the apostrophe used to escape out of the parameter, as shown in Listing 15.

EXEC Production.VulnerableSearch 'Bike'' OR 1 = 1--';

Listing 15: An injection attack on the VulnerableSearch stored procedure.

By using a dynamic stored procedure with @Filter limited to 50 characters, we do gain a little bit of protection because an attacker could not craft a very long string. However, a "drop table" statement isn't that long.

Another form of fundamentally broken dynamic SQL statement is the list search. It is inevitable that a developer will want to create something like an array in SQL, to pass in a list and search for rows that match any value in that list. If they stumble upon Erland Sommerskog's excellent essay on the topic (<http://www.sommarskog.se/arrays-in-sql.html>) then, hopefully, these developers will use one of the proper methods for simulating an array. Unfortunately, there are certain solutions that are more problematic. Listing 16 shows an example of one.

CREATE PROCEDURE Production.NotSecure

@NameList NVARCHAR(MAX)

AS

DECLARE @sql NVARCHAR(MAX);

SET @sql = N'select \* from Production.ProductSubcategory where Name IN ('

+ @NameList + N');';

EXEC sp\_executesql @sql, N'@NameList nvarchar(max)', @NameList;

GO

--try it out

EXEC Production.NotSecure N'''Mountain Bikes'', ''Road Bikes''';

Listing 16: Bad list search using dynamic SQL and **sp\_executesql**.

This procedure uses sp\_executesql, which takes three primary parameters:

* **@statement** – the SQL statement to run
* **@params** – an nvarchar string containing a list of the set of parameters the query will use
* a series of **@param(x)** – a comma-delimited set of the actual parameters used.

The idea is that, by entering the parameters directly into the statement, we allow SQL Server to parameterize these variables correctly, ensuring that users cannot escape out of the query. Going back to correctly parameterized queries through ASP.NET, they all use sp\_executesql and the variables are all "inside" the SQL query.

Unlike the correctly parameterized queries that ASP.NET generated before, this particular way of using sp\_executesql is not safe. The reason is that, even though we pass @NameList as a query parameter, we actually make use of @NameList before sp\_executesql has a chance to run. The sp\_executesql procedure will not throw an exception or generate any type of error if a parameter is listed that is not actually used in the SQL query, meaning that developers and DBAs must remain vigilant when it comes to using sp\_executesql correctly. The consequences are potentially devastating.

EXEC Production.NotSecure N'''MountainBikes'') OR 1 = 1--';

Listing 17: Injection attack on bad list search.

In this case, we can pass in a specially crafted string and get around sp\_executesql's built-in SQL injection protection mechanism. On the web form, we would need to double up the apostrophe after "MountainBikes" and add a closed parenthesis before using our "OR 1=1" statement. Getting the correct syntax may take a few tries for an attacker who does not have access to source code, but it is fundamentally no more difficult than the basic examples we have covered already. Once the attacker has the correct syntax, the results are the same, sp\_executesql or no sp\_executesql.

The fact that we were able to exploit an SQL injection vulnerability in a dynamic SQL statement that uses sp\_executesql, however, is certainly not the fault of the system stored procedure; rather, it is the fault of the developer who did not use sp\_executesql correctly in the first place. The sp\_executesql system stored procedure translates parameters and makes them safe for use, but we did not actually pass any parameters into the command; instead, we translated the parameters first and passed in a basic string of text to the sp\_executesql procedure. Listing 18 shows a more appropriate use of sp\_executesql.

CREATE PROCEDURE Production.InvulnerableSearch @Filter VARCHAR(50)

AS

DECLARE @sql NVARCHAR(250);

SET @sql = N'select \* from Production.ProductSubcategory where Name like ''%@Filter%'';';

EXEC sp\_executesql @sql, N'@Filter varchar(50)', @Filter;

GO

Listing 18: Good list search using dynamic SQL and sp\_executesql.

This particular dynamic SQL search, which matches Production.VulnerableSearch's intention and result set, is invulnerable to SQL injection attack. Note that, unlike our previous attempts at using sp\_executesql, @Filter is entirely "within" the @sql variable. sp\_executesql translates the value of @Filter and makes it safe from SQL injection *before* executing the dynamic SQL statement, therefore offering the same level of SQL injection protection as a static stored procedure or parameterized ad hoc query.

**Other Forms of Defense**

At this point, we know how to create code that is invulnerable to SQL injection. We learned that the correct way to handle application-level SQL calls is through parameterized queries. In .NET languages like C# and VB.NET, we have access to easy-to-use classes and methods, some of which we discussed in this article. Other web frameworks and programming languages have their own parameterization libraries, so if you're using something other than the Microsoft stack, use whatever equivalent parameterization functionality is available.

In addition to parameterization at the application level, we also learned how to secure stored procedures in SQL Server. Static stored procedures are already safe from SQL injection, and we can protect against SQL injection in dynamic stored procedures through proper use of sp\_executesql.

Before wrapping up this article, I would like to touch upon a few additional areas of interest with regard to SQL injection. It would be easy to write entire chapters on these, but I will have to settle for a few sentences on each topic.

**QUOTENAME and REPLACE instead of sp\_executesql**

Kimberly Tripp has a great set of blog posts on how to protect against SQL injection attacks by making appropriate use of the built-in QUOTENAME and REPLACE functions when calling the exec command. She argues in favor of using these instead of sp\_executesql because it can perform significantly better in certain circumstances. By contrast, Aaron Bertrand prefers sp\_executesql.

* [**Little Bobby Tables, SQL Injection and EXECUTE AS**](http://tinyurl.com/pwckpkz)
* [**EXEC and sp\_executesql – how are they different?**](http://tinyurl.com/on68eqj)
* [**Bad Habits to Kick: Using EXEC() instead of sp\_executesql**](http://tinyurl.com/p5ujw9x)

In this particular debate, I side with Bertrand, which is why I tended to use sp\_executesql above instead of exec with QUOTENAME and REPLACE. Although sp\_executesql is often slower than simply running exec with appropriate use of QUOTENAME and REPLACE, it is also a lot easier to get everything right. This is especially true for application developers writing stored procedures that use dynamic SQL. In that case, I would not automatically trust them (or even myself!) to get it right and, instead, would focus on the easier safe method, at least until performance simply is not good enough.

**Appropriate permissions**

Throughout this article, I have assumed that the account running these stored procedures and ad hoc SQL statements has some hefty rights, probably db\_owner, and maybe even sysadmin. Sadly, this is usually a safe assumption.

If a procedure runs a SELECT statement that only hits three tables in the AdventureWorks database, it does not need permission to insert records, create user accounts, run xp\_cmdshell, or view the INFORMATION\_SCHEMA or sys schemas! Erland Sommarskog's outstanding essay, [Giving Permissions through Stored Procedures](http://www.sommarskog.se/grantperm.html), describes in detail various methods available for granting appropriate permissions for stored procedures (see also, the previously referenced Kimberly Tripp blog post on the EXECUTE AS functionality in SQL Server, and how to use that to allow only limited access to SQL stored procedures).

Even if the resources are not available to create appropriate permissions and signed certificates for each stored procedure, at least create limited-functionality accounts and run stored procedures through those accounts. Reader and editor logins that use the sp\_datareader and sp\_datawriter security roles, respectively, would at least protect against certain shenanigans, such as attempts to drop tables, or to use what a developer intends to be a SELECT statement to insert or delete records.

**Automated tools**

A number of tools can help discover possible weaknesses in code, and so prevent attacks. One of my favorites is sqlmap (<http://sqlmap.org/>), which allows SQL injection attack attempts against a number of database vendors' products, not just SQL Server. It also lets the user perform advanced SQL injection techniques not covered in this article, such as timing-based injection attacks.

In addition to those features, it can tie into Metasploit ([http://metasploit.com](http://metasploit.com/)), an outstanding penetration-testing suite. Inside Metasploit, there are SQL Server-specific modules such as one that tries to guess a SQL Server instance's sa password. If it is successful, the module automatically tries to create a Metasploit shell, giving the attacker full access to the database server. Both tools come with thorough documentation and can be automated for enterprise-level penetration tests.

**Web application firewall**

Web application firewalls (WAFs) are a hot topic in the security world at present. Most of them claim to protect against SQL injection attacks, and they often do a reasonable job. Given what we've discussed in this article, there are many ways to sneak in injection code, so I would not recommend using a WAF as the only line of defense.

Even if your web application firewall does block all SQL injection attempts perfectly, does it fail open or fail closed? In other words, in the event that somebody targets a denial of service attack on the web application firewall and it ceases to function, will it allow all traffic to go on through, or will it effectively shut down the website by failing closed? If the product fails open, then an attacker could still perform SQL injection, although the attacker would need to be considerably more sophisticated than if there was no WAF.

WAF products are worth investigating, if the price is right, as they can be a valuable addition to a security infrastructure. They are not, however, the best line of defense against SQL injection: good development practices and parameterized queries are.

**Additional Resources**

Thanks to the prevalence of SQL injection attacks, finding examples of attacks is a simple exercise. As of this article's publication, a few recent attacks include:

* Stratfor – <http://tinyurl.com/StratforSQLi>
* IRC Federal – <http://tinyurl.com/IRCFederalSQLi>
* Sony – <http://tinyurl.com/SonySQLi>
* Arthur Hicken of Parasoft – a curated list of high-profile SQL injection attacks, located at <http://codecurmudgeon.com/wp/sql-injection-hall-of-shame/>.

In terms of thwarting SQL injection, I recommend the following additional resources:

* **Bobby Tables** – a compendium of ways to parameterize queries in different programming languages, including technologies outside of ASP.Net and SQL Server <http://bobby-tables.com/>
* **Microsoft's best practice guidance on SQL** <http://tinyurl.com/nqhgj3g>
* **Erland Sommarskog's essay on dynamic SQL – includes a primer on SQL injection** <http://tinyurl.com/pxpgc82>