MS_logo_KQuick Security Reference:   
SQL Injection

December 17, 2009

For the latest information, please see <http://www.microsoft.com/sdl>.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form, by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft, SharePoint, SQL Server, Visio, Visual Basic, Visual C#, Visual C++, Visual Studio, the Visual Studio logo, Win32, Windows, Windows Mobile, Windows Server, and Windows Vista are trademarks of the Microsoft group of companies.

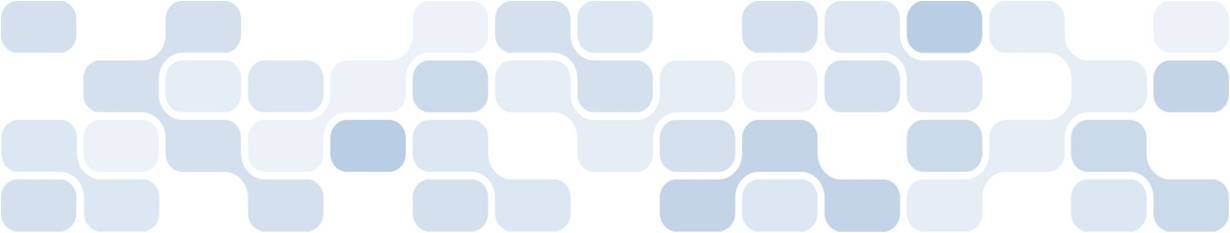
The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Table of Contents

[OVERVIEW 3](#_Toc245695817)

[WHAT ARE PERSONAS 3](#_Toc245695818)

[Business Decision Maker 4](#_Toc245695819)

[Architect 4](#_Toc245695820)

[Developer 4](#_Toc245695821)

[Tester/QA 4](#_Toc245695822)

[UNDERSTANDING SQL INJECTION FOR THE BUSINESS DECISION MAKER 4](#_Toc245695823)

[Risk 4](#_Toc245695824)

[Business Impact 5](#_Toc245695825)

[Fixing the Code 6](#_Toc245695826)

[Resources and Training for Business Decision Makers 7](#_Toc245695827)

[UNDERSTANDING SQL INJECTION FOR THE ARCHITECT/PM 7](#_Toc245695828)

[Identifying the Problem 7](#_Toc245695829)

[Common SQL Injection Attacks 7](#_Toc245695830)

[Designing a Fix 7](#_Toc245695831)

[Tools for Designing Software That Prevents SQL Injection Vulnerabilities 9](#_Toc245695832)

[Resources and Training for Architects/PMs 9](#_Toc245695833)

[UNDERSTANDING SQL INJECTION ATTACKS FOR THE DEVELOPER 9](#_Toc245695834)

[Example of SQL Injection 9](#_Toc245695835)

[Example of a SQL Injection by Truncation Attack 10](#_Toc245695836)

[Writing Secure Code 12](#_Toc245695837)

[Constrain Input 12](#_Toc245695838)

[Use Parameterized SQL Queries 13](#_Toc245695839)

[Use Proper Escaping Techniques to Handle Special Input Characters 14](#_Toc245695840)

[Calculate Buffer Lengths Properly 15](#_Toc245695841)

[Additional Considerations 16](#_Toc245695842)

[Use a Least-Privileged Database Account 16](#_Toc245695843)

[Avoid Disclosing Detailed Error Information 16](#_Toc245695844)

[Tools and Libraries 17](#_Toc245695845)

[Resources and Training for Developers 17](#_Toc245695846)

[UNDERSTANDING SQL INJECTION VULNERABILITIES FOR THE TESTER/QA 17](#_Toc245695847)

[Map Out the Site and Its Functionality 17](#_Toc245695848)

[Start Testing and Pay Attention to the Output 18](#_Toc245695849)

[Techniques for Finding Various Types of SQL Injection Vulnerabilities 18](#_Toc245695850)

[Code Reviews 18](#_Toc245695851)

[Building Improvements into Black Box Testing 19](#_Toc245695852)

[Tools You Can Use 19](#_Toc245695853)

[Resources and Training for Testers 19](#_Toc245695854)

[THE MICROSOFT SDL AND PREVENTING SQL INJECTION ATTACKS 20](#_Toc245695855)

[Long-Term Solutions 20](#_Toc245695856)

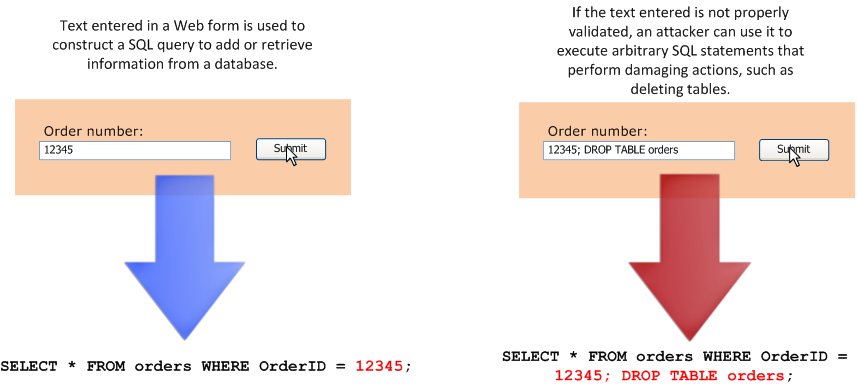
[CONCLUSION 21](#_Toc245695857)

[ACKNOWLEDGMENTS 21](#_Toc245695858)

# OVERVIEW

SQL injection attacks have become one of the most common and dangerous Web application security issues on the Internet. *SQL injection vulnerabilities* occur when an application takes user content data and uses it to construct SQL (Structured Query Language) statements without first properly validating or sanitizing that content. *SQL injection attacks* take advantage of SQL injection vulnerabilities to steal sensitive data from the database, modify or destroy the stolen data, execute administrative commands on the database, or in some cases take control of the whole machine. In recent years, SQL injection attacks have been used to store malware in databases and then distribute them through Web sites that are hosted on these compromised databases.

SQL injection attacks can occur using a variety of techniques. The diagram below describes a simple attack to drop a SQL database table from a Web page that accepts user input from a text box.



*Figure 1. Example of a basic SQL injection attack*

This Quick Security Reference (QSR) paper can be used by each member of your engineering organization to gain a more thorough understanding of how to address SQL injection attacks. By better understanding these vulnerabilities, you can more easily and efficiently deal with existing issues and implement ongoing solutions that better protect your software, Web sites, and users.

# WHAT ARE PERSONAS

Every person from the business decision maker, to the architect, to the developer, and tester/quality assurance (QA) must play a role in addressing this issue. However, each person must approach the problem from a different perspective to ensure that you have a clear plan for fixing the SQL injection issue, successfully verifying your solution, and mitigating the problem in the future.

This paper uses four basic personas in an engineering organization—business decision maker, architect, developer, and tester/QA. The goal of this paper is to address the key questions that each persona might ask regarding a SQL injection attack and to provide direction for each persona on how to address the issue. The personas are defined in the following paragraphs.

## Business Decision Maker

* In charge of delivering the development project from start to finish, on time, and within budget.
* Aware of the fact that security is a customer-perception and cost-management issue.
* Needs to understand security issues at a high level to make overall project planning decisions, including budget and justification of time and resource investments toward security development.

## Architect

* In charge of the technical design of the development project.
* Cares about security from an overall integrity standpoint.
* Understands security concepts in enough detail to avoid creating a blueprint that would result in security issues.
* Responsible for creating the core principles that developers use to implement the functional goals of the project, managing awareness of security issues as they relate to the specific solution, and tracking and verifying fixes to potentially costly security mistakes in code leading up to release.

## Developer

* In charge of taking the architectural blueprints for the development project and implementing them in code.
* Fixes security issues from coding errors.
* Understands basic secure coding best practices in detail and gathers supporting technical content, such as approved libraries, to ease the ability to write secure code correctly and consistently.

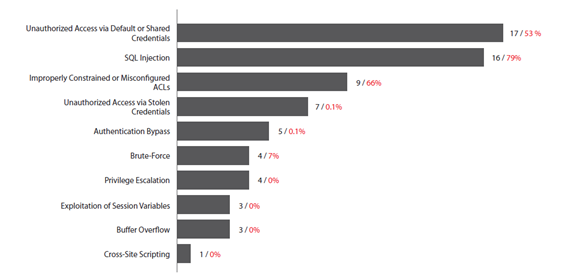
## Tester/QA

* In charge of acting as the quality gatekeeper and managing the last step before a development project is completed.
* Ensures that any security issues that may have slipped through previous steps are caught before the application is released.
* Executes security testing (different than functional testing) to attempt to make the application do things that it was not designed to do. If successful, these tests may represent potential vulnerabilities.
* Maintains a detailed understanding of attack techniques and a bit of the creative “attacker’s mindset.”

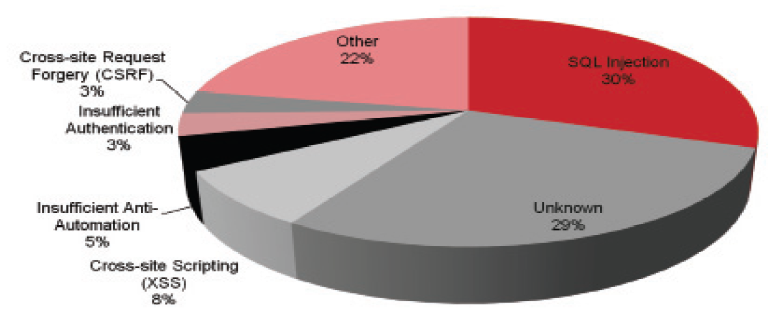
# UNDERSTANDING SQL INJECTION FOR THE BUSINESS DECISION MAKER

## Risk

SQL injection attacks have become the favorite way to attack a database and steal sensitive information. They have gained popularity and effectiveness because this type of attack can effectively compromise data layers despite the existence of firewalls and intrusion detection systems. According to Verizon Business, in 2008 SQL injection attacks ranked first in the number of records compromised—79 percent of the aggregate 285 million.[[1]](#footnote-1)

*Figure 2. Types of hacking by number of breaches (black) and percent of records (red)*

According to breach.com, SQL injection attacks were the most common type of security incident resulting in security breaches in 2008.[[2]](#footnote-2)

*Figure 3. Breakdown of security breaches for 2008 by type of security incident*

In recent years, attackers have been using SQL injection attacks as a way to distribute malware onto SQL servers. The risk of SQL injection attacks today is much greater than a few years ago because the probability of an attacker exploiting a SQL injection vulnerability is more likely.

## Business Impact

There have been several recent high-profile SQL injection attacks that had a direct effect on the day-to-day operations, customer-facing solutions, and business practices of companies.

Some high-profile SQL injection attacks include:

* “Federal authorities have charged a previously indicted hacker with breaching additional corporate computers and stealing data for at least 130 million credit and debit cards, the biggest identity theft case ever prosecuted in the United States. Albert "Segvec" Gonzalez and two unnamed Russians were indicted on Monday for attacks that hit credit card processor Heartland Payment Systems, retailers 7-Eleven and Hannaford Brothers, and two unidentified companies. Documents filed in US District Court in Newark, New Jersey claim that ... the trio used SQL injection attacks to install sniffer software on the companies' servers to intercept credit card data as it was being processed. ... The breach has proved to be a major embarrassment for Heartland, which ... has so far allocated $12.6m to cover costs stemming from the loss of sensitive card-holder data.”[[3]](#footnote-3)
* “In what could be the largest data security breach to date, MasterCard International on Friday said information on more than 40 million credit cards may have been stolen. The breach occurred at CardSystems Solutions in Tucson, Ariz., a third-party processor of payment data, according to a MasterCard statement.”[[4]](#footnote-4)
* “In September 2004, a hacker exploited the failures set forth… by using an SQL injection attack on respondent’s web application and website to install common hacking programs on computers on respondent’s computer network. The programs were set up to collect and transmit magnetic stripe data stored on the network to computers located outside the network every four days, beginning in November 2004. As a result, the hacker obtained unauthorized access to magnetic stripe data for tens of millions of credit and debit cards.”[[5]](#footnote-5)
* “Malware found on an internal database may have allowed spammers to steal names, addresses, phone numbers, and email addresses from as many as 6.3 million customers of TD Ameritrade, the brokerage firm revealed today.”[[6]](#footnote-6) “Carl Banzhof, vice president and chief technology evangelist at McAfee, told SCMagazineUS.com today that the cyber thieves likely used SQL injection tactics to infiltrate the database, harvesting email addresses.”[[7]](#footnote-7)

## Fixing the Code

SQL injection attacks are most effectively addressed in the design phase of your software life cycle by looking for ways to prevent them based on your solution and architectural design. Most commonly, SQL injection vulnerabilities are a result of coding vulnerabilities during the Implementation/Development phase and will likely require code changes by the same team who authored and are most familiar with the code. Ongoing security education of your developers should be a critical piece of your life cycle to equip them with the knowledge needed to effectively identify, address, and avoid these vulnerabilities in their code. Additionally, SQL injection attacks should be used in the Verification/QA/Test phase. Using SQL injection vulnerabilities will help you exercise your Web application to ensure mitigations and protections are in place. The QA/Test team should be the most effective team at uncovering SQL injection vulnerabilities prior to releasing a solution.

## Resources and Training for Business Decision Makers

* [Web Hacking Incidents Database 2008](http://www.breach.com/resources/whitepapers/downloads/WP_WebHackingIncidents_2008.pdf)
* SQL Injection topic in [SQL Books Online](http://msdn.microsoft.com/en-us/library/ms161953.aspx)
* [*19 Deadly Sins of Software Security*](http://www.amazon.com/Deadly-Sins-Software-Security-Programming/dp/0072260858/ref=sr_1_1?ie=UTF8&s=books&qid=1256159028&sr=8-1) by Michael Howard, David LeBlanc, and John Viega

# UNDERSTANDING SQL INJECTION FOR THE ARCHITECT/PM

## Identifying the Problem

SQL injection vulnerabilities are generally categorized into three different types—first order, second order, and truncation vulnerabilities.

* **First-order SQL injection** vulnerabilities are the easiest to exploit. In this type of exploit, a page constructs SQL statements based on the attacker-supplied data. These are the most common types of attacks.
* In **second-order SQL injection** attacks, an attacker submits hostile data through a Web page that stores it in a file, database, or other data store, and depends on some other Web page or mid-tier application to use that hostile data in the construction of a SQL statement.
* In **SQL injections by truncation**, the attacker uses varying buffer lengths to truncate a delimited string and uses another input value to inject his commands of choice into a dynamically constructed SQL statement.

## Common SQL Injection Attacks

The common SQL injection attack exploits vulnerabilities found in Web pages. The attack is executed by injecting SQL statements as part of data input to Web pages. A vulnerable Web page sends these SQL statements to a database server, which in turn executes them. These SQL statements are executed under the context of the account that the Web page uses to connect to Microsoft SQL Server®.

There are two common exploit payloads types that attackers use:

1. **Use UNION SELECT statement** to steal sensitive information from the database server.
2. **Batch multiple SQL commands** either to tamper with data, destroy data, or execute SQL commands of their choice on the database server.

## Designing a Fix

The Microsoft Security Development Lifecycle (SDL) requires all Web applications (and any type of application that executes SQL statements on SQL Server) to implement a clearly defined set of defenses against SQL injection attacks. Whether you have adopted the SDL in your development process or not, the following defenses against SQL injection attacks are vital to consider when designing your Web application:

* **All input to the application from a user, a component, or another program should be validated.** This helps to ensure that the input is free from characters that cause SQL injection attacks. It not only helps to protect against SQL injection attacks but also other attacks, such as XSS attacks and buffer overflows.
* **Use parameterized SQL queries.** Applications accessing a database must do so using only parameterized queries. Creating dynamic queries using string concatenation potentially allows an attacker to execute an arbitrary query through the application.
* **Use stored procedures.** Using stored procedures helps to mitigate the SQL injection threat to a great extent because type checking is available for parameters. If the attacker supplies input that does not match the type constraints, the stored procedures throw an exception. In the vast majority of the cases, this should be properly handled within the application. However, if the stored procedures perform string manipulation in the code and then execute that query using the "exec@sql" construct, incorrect handling of user input can produce the same SQL injection vulnerability as would be seen at the application layer.

**Note:** Stored procedures by themselves do not remove SQL injection vulnerabilities. They only raise the bar on the attacker by hiding much of the underlying database schema.

* **Use SQL execute-only permission.** This is a defense-in-depth method. This defense assumes the attacker has successfully found a SQL injection bug in your code. Now what? Thankfully, this defense stops almost every attack dead in its tracks. From the SDL, “Only grant ‘execute' permission on all stored procedures, and grant that permission only for the application domain group. Ensure that this group is granted execute permissions only on your stored procedures. Do not grant any other permission on your database to any other user or group.” This is a great defense because if the attacker attempts to access any other database object other than through a stored procedure (you can use views also), the underlying database permissions model prevents the attack by denying the attacker access. In general, database applications should be using a low-privileged account that has the minimum permissions required to execute the statements submitted to SQL Server. One should never use high-privileged accounts like dbo or sysadmin. If high-privileged operations need to be performed, then wrap those operations in a stored procedure and sign that stored procedure with a certificate that has the required high privileges and grant execute permission on the stored procedure.

When implemented, the first SQL injection requirement ensures that an application is secure by design, and thus protected from SQL injection attacks. The other two requirements provide security by default defenses to mitigate other attacks if existing protections were to fail.[[8]](#footnote-8)

There are few additional requirements that one must follow:

* **Use proper escaping for SQL identifiers and Data Definition Language (DDL) statements.** There may be times when SQL identifiers (table names, column names, and so on) might be constructed dynamically, or there may be some DDL statements for which one cannot use parameterized SQL statements. In these situations, one should use proper escaping techniques to mitigate SQL injection attacks.
* **Handle buffer lengths properly.** If dynamic SQL statements are being constructed for the type of scenarios mentioned earlier, buffer lengths need to be handled properly to prevent SQL injections through truncation. This topic is discussed at length in the MSDN article [New SQL Truncation Attacks and How to Avoid Them](http://msdn.microsoft.com/en-us/magazine/cc163523.aspx).

## Tools for Designing Software That Prevents SQL Injection Vulnerabilities

The [Microsoft SDL Threat Modeling Tool](http://msdn.microsoft.com/en-us/security/dd206731.aspx) (free download) can be helpful in reviewing the implementation of the aforementioned rules during product design and in identifying potential SQL injection threats you need to mitigate when you start writing code.

## Resources and Training for Architects/PMs

* [SDL Process Guidance, Implementation Phase](http://msdn.microsoft.com/en-us/library/cc307416.aspx)
* [MSDN Patterns and Practices: How To: Protect From SQL Injection in ASP.NET](http://msdn.microsoft.com/en-us/library/ms998271.aspx)
* SQL Injection topic in [SQL Books Online](http://msdn.microsoft.com/en-us/library/ms161953.aspx)

* *[19 Deadly Sins of Software Security](http://www.amazon.com/Deadly-Sins-Software-Security-Programming/dp/0072260858/ref=sr_1_1?ie=UTF8&s=books&qid=1256159028&sr=8-1)* by Howard, LeBlanc, and Viega
* [*Writing Secure Code* (Second Edition)](http://www.amazon.com/Writing-Secure-Second-Michael-Howard/dp/0735617228/ref=sr_1_1?ie=UTF8&s=books&qid=1256162128&sr=8-1) by Howard and LeBlanc

# UNDERSTANDING SQL INJECTION ATTACKS FOR THE DEVELOPER

A successful SQL injection attack enables a malicious user to execute commands in your application’s database by using the privileges granted to your application’s login. The problem is more severe if your application uses an over-privileged account to connect to the database. For example, if your application’s login has privileges to eliminate a database, then without adequate safeguards an attacker might be able to perform the operations described in this section.

### Example of SQL Injection

Consider the following ASP.NET code that retrieves first name and last name of an author based on her Social Security Number (SSN) that is supposed to be in the form nnn-nn-nnnn:

// Use dynamic SQL

SqlDataAdapter myCommand = new SqlDataAdapter(

"SELECT au\_lname, au\_fname FROM authors WHERE au\_id = '" +

SSN.Text + "'", myConnection);

The developer's intention was that when the code runs, it inserts the user’s input and generates a SQL statement similar to the following:

SELECT au\_lname, au\_fname FROM authors WHERE au\_id = '172-32-9999'

However, an attacker can supply the SSN number as, 172-32-9999' ; DROP DATABASE pubs***,*** in which case the statement becomes:

SELECT au\_lname, au\_fname FROM authors WHERE au\_id = '172-32-9999'; DROP DATABASE pubs --'

In this case, the **'** (single quotation mark) character contained in the rogue input terminates the current string literal in the SQL statement. It closes the current statement only if the following parsed token does not make sense as a continuation of the current statement, but does make sense as the start of a new statement. As a result, the single quotation mark character of the rogue input results in the following statement:

SELECT au\_lname, au\_fname FROM authors WHERE au\_id = '172-32-9999'

The ; (semicolon) character tells SQL that this is the end of the current statement, which is then followed by malicious SQL code.

; DROP DATABASE pubs

**Note:** The semicolon is not necessarily required to separate SQL statements. This is dependent on the vendor or implementation, but Microsoft SQL Server does not require them. For example, SQL Server parses the following as two separate statements:

SELECT \* FROM MyTable DELETE FROM MyTable

Finally, the -- (double dash) sequence of characters is a SQL comment that tells SQL to ignore the rest of the text. In this case, SQL ignores the closing ' (single quotation mark) character, which would otherwise cause a SQL parser error.

--'

### Example of a SQL Injection by Truncation Attack

Consider the following stored procedure that alters a SQL login’s password based on the current password.

create procedure sys.sp\_password

@old nvarchar(25), -- the old (current) password

@new nvarchar(25), -- the new password

@loginame nvarchar(25) -- user to change password on

as

-- DECLARE VARIABLES --

set nocount on

declare @exec\_stmt nvarchar(4000)

declare @escaped\_oldpw nvarchar(25)

declare @escaped\_newpw nvarchar(25)

set @escaped\_oldpw = replace(@old, '''', '''''')

set @escaped\_newpw = replace(@new, '''', '''''')

set @exec\_stmt = N'alter login ' + quotename(@loginame) +

' with password = ''' + @escaped\_newpw + N''' old\_password = ''' +

@escaped\_old + N'''

exec (@exec\_stmt)

if @@error <> 0

return (1)

-- RETURN SUCCESS --

return (0) -- sp\_password

The **ALTER LOGIN** statement cannot be parameterized, so the only option is to construct the query dynamically. In this situation, one needs to escape single quotes in data values (password values) and escape right square brackets in SQL identifiers (login name) to mitigate SQL injection vulnerabilities. The **Quotename()** function in TSQL automatically escapes right square brackets and delimits the identifier with square brackets. By following these rules for escaping values in passwords and login names, this code is not vulnerable to SQL injection.

For example, if someone provides the following values:

@loginname => sa

@old =>dummyold

@new =>dummy'; drop table customers --

Then the final statement becomes:

Alter login [sa] with password='dummy''; drop table customers—' with password='dummyold'

Since the code has escaped single quotes in the @new variable, this attack doesn’t work as the entire string dummy'; drop table customers is treated as the string token in the statement.

Similar attacks using the @login or @old variables wouldn’t work either, so the code seems to be safe against SQL injections, which is what the developer tried to address.

However, if someone passes 1234…[24] ' for the @new variable, the final command becomes:

Alter login [sa] with password='123…[24] '' with password='dummyold'

If you look closely at the stored procedure shown previously, you will notice that the @escaped\_newpw variable can only hold 25 characters, but a **REPLACE** function on the @new variable of 25 characters can return 50 characters when all the characters are single quotes. All versions of SQL Server silently truncate the data if the variable doesn't have big enough buffers. This provides attackers an opportunity to escape out of the string token and inject commands of their choice through some other variables.

For example, an attacker could provide the following values and bypass the SQL injection mitigation put in place:

@loginname => sa

@old =>; drop table customers --

@new =>1234…[24]'

Then the final statement becomes:

Alter login [sa] with password='123…[24]'' with password=';drop table customers --'

Effectively the attacker was able to execute two statements:

1. Alter login [sa] with password='123…[24]''password='
2. drop table customers --'

While this example uses TSQL code, similar problems are possible in C/C++ code as well. Most of the time, you can avoid construction of dynamic SQL by making use of parameterized DML statements or stored procedure calls, but when you need to construct the statements dynamically (for DDL operations), you need to be careful of first-order/second-order and SQL injection by truncation issues.

## Writing Secure Code

Now that some background and examples of how an attacker might go about carrying out SQL injection attacks have been covered, it’s time to take a look at how writing secure code can mitigate these attacks.

### Constrain Input

You should validate all input to your applications for type, length, format, and range. By constraining the input used in your data access queries, you can protect your application from SQL injection and other attacks. When constraining input, it is a good practice to create a list of acceptable characters and use regular expressions to reject characters that are not on the list. The potential risk associated with using a list of unacceptable characters is that it is always possible to overlook an unacceptable character when defining the list. Also, an unacceptable character can be represented in an alternate format to pass validation.

In the first example, since the input is an SSN in the form nnn-nn-nnnn, one can write the following regular expression validation to validate the format of the input:

if (Regex.IsMatch(Request.QueryString["SSN"], "^\d{3}-\d{2}-\d{4}$"))

{

// access the database

}

else

{

// handle the bad input

}

### Use Parameterized SQL Queries

The best way to mitigate SQL injection attacks is to use parameterized SQL queries. In the previous example, instead of constructing the dynamic SQL statement based on the user-supplied data, you can write the following code that makes use of parameterized SQL:

using System.Data;

using System.Data.SqlClient;

using (SqlConnection connection = new SqlConnection(connectionString))

{

DataSet userDataset = new DataSet();

SqlDataAdapter myDataAdapter = new SqlDataAdapter(

"SELECT au\_lname, au\_fname FROM Authors WHERE au\_id = @au\_id",

connection);

myCommand.SelectCommand.Parameters.Add("@au\_id", SqlDbType.VarChar, 11);

myCommand.SelectCommand.Parameters["@au\_id"].Value = Request.QueryString["SSN"];

myDataAdapter.Fill(userDataset);

}

A better solution is to wrap up the underlying commands in a stored procedure and execute that stored procedure from the application. The main advantage of this design is that the account used by the Web application only needs to have execute permission on the stored procedure rather than a **SELECT** statement on the underlying table. This way, in the event of a SQL injection vulnerability in some other part of the application, the attacker cannot directly manipulate the underlying tables.

Create procedure sp\_GetAuthorDetails

@ssn varchar(11)

BEGIN

Select au\_lname, au\_fname FROM Authors WHERE au\_id = @ssn

END

With such a stored procedure definition, the application code can be modified as follows to address SQL injection vulnerabilities:

using System.Data;

using System.Data.SqlClient;

using (SqlConnection connection = new SqlConnection(connectionString))

{

DataSet userDataset = new DataSet();

SqlDataAdapter myCommand = new SqlDataAdapter(

" sp\_GetAuthorDetails", connection);

myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;

myCommand.SelectCommand.Parameters.Add("@au\_id", SqlDbType.VarChar, 11);

myCommand.SelectCommand.Parameters["@au\_id"].Value = Request.QueryString["SSN"];

myCommand.Fill(userDataset);

}

### Use Proper Escaping Techniques to Handle Special Input Characters

In situations where parameterized SQL cannot be used and you are forced to use dynamic SQL instead, you need to safeguard against input characters that have special meaning to SQL Server, such as the single quote character for string literals and right square brackets for SQL object names. If not handled, special characters, such as the single quote character in the input, can be utilized to cause SQL injection.

***Note:*** *Special input characters pose a threat only with dynamic SQL and not when using parameterized SQL.*

Escape routines add an escape character to characters that have special meaning to SQL Server, thereby making them harmless. The escape techniques for SQL identifiers (object names) and string literals are highlighted in the following text.

*Delimiting and Escaping SQL Identifiers*

* Use delimited identifiers:
  + When reserved words are used for object names.
  + When you are using characters that are not listed as qualified identifiers.
  + When you are constructing a DDL statement that accepts variable SQL identifiers.
* Double quotes can be used to delimit identifiers based on where QUOTED\_IDENTIFIER is ON or OFF, so it is not a good practice to use double quotes as the delimiters for SQL identifiers.
* Never use single quotes to delimit identifiers.
* Always use square brackets ([ and ]) to delimit identifiers.
* Double up all occurrences of right square brackets (]) in the object name.

*Delimiting and Escaping String Data*

* Double quotes can be used to delimit literals based on where QUOTED\_IDENTIFIER is OFF or ON, so it is not a good practice to use double quotes as the delimiters for string literals.
* Always use single quotes to delimit literals.
* Double up all occurrences of single quotes in the string literals.

The earlier code snippet is a good example of using these proper escaping and delimiting techniques. The SQL identifier is delimited with square brackets and all right square brackets are escaped. Similarly, both password string literals are delimited with single quotes and escaped single quotes. However, there is one problem with the snippet, which is discussed in the next section.

create procedure sys.sp\_password

@old nvarchar(25), -- the old (current) password

@new nvarchar(25), -- the new password

@loginame nvarchar(25) -- user to change password on

as

-- DECLARE VARIABLES --

set nocount on

declare @exec\_stmt nvarchar(4000)

declare @escaped\_oldpw nvarchar(25)

declare @escaped\_newpw nvarchar(25)

set @escaped\_oldpw = replace(@old, '''','''''')

set @escaped\_newpw = replace(@new, '''','''''')

set @exec\_stmt = N'alter login [' + replace(@loginame,'] ',']]') +

‘] with password = ''' + @escaped\_newpw + N''' old\_password = ''' +

@escaped\_old + N''''

exec (@exec\_stmt)

if @@error <> 0

return (1)

-- RETURN SUCCESS –

return (0) -- sp\_password

### Calculate Buffer Lengths Properly

In situations where you need to construct SQL statements, apart from handling special characters, you should also handle string lengths properly. SQL injection by truncation attacks are more relevant for the code developed in C/C++ and TSQL and do not exist in ASP or ASP.NET.

Here is an example of how to fix the previous code snippet that has a SQL injection by truncation vulnerability:

create procedure sys.sp\_password

@old nvarchar(25), -- the old (current) password

@new nvarchar(25), -- the new password

@loginame nvarchar(25) -- user to change password on

as

-- DECLARE VARIABLES --

set nocount on

declare @exec\_stmt nvarchar(4000)

declare @escaped\_oldpw nvarchar(50)

declare @escaped\_newpw nvarchar(50)

set @escaped\_oldpw = replace(@old, '''','''''')

set @escaped\_newpw = replace(@new, '''','''''')

set @exec\_stmt = N'alter login [' + replace(@loginame,']',']]') +

'] with password = ''' + @escaped\_newpw + N''' old\_password = ''' +

@escaped\_old + N''''

exec (@exec\_stmt)

if @@error <> 0

return (1)

-- RETURN SUCCESS –

return (0) -- sp\_password

The fix is quite simple—change the buffer sizes of @escaped\_newpw and @escaped\_oldpw to nvarchar(50). In the event that all characters in the password field are single quotes, the **REPLACE** function return doubles up all single quotes and returns (25\*2) characters. So by increasing the buffer sizes to 50, the truncation vulnerability is addressed. If the attacker supplies the following values:

@loginname => sa

@old =>; drop table customers --

@new =>1234…[24]'

Then the final statement becomes only one statement and all the values are treated as string literals.

Alter login [sa] with password='123…[24]''' with password=';drop table customers --'

## Additional Considerations

Along with the aforementioned coding practices for mitigating SQL injection attacks, you can consider the following defense-in-depth countermeasures to prevent SQL injection:

* Use a least-privileged database account.
* Avoid disclosing error information.

### Use a Least-Privileged Database Account

Your application should connect to the database using a least-privileged account. If you use Windows® authentication to connect, the Windows account should be least-privileged from an operating system perspective and should have limited privileges and limited ability to access Windows resources. Additionally, whether you use Windows authentication and/or SQL authentication, the corresponding SQL Server login should be restricted by permissions in the database. If your application only executes stored procedures, then that account should be restricted to only execute on those stored procedures and nothing more. Similarly, if you execute SQL statements other than stored procedure calls, then you should only be granting the minimum permissions required to execute those statements. This design principle acts as a defense-in-depth technique to limit the damage potential of a SQL injection vulnerability in the application.

### Avoid Disclosing Detailed Error Information

Use structured exception handling to catch errors and prevent them from propagating back to the client. Log detailed error information locally, but return limited error details to the client.

If errors occur while the user is connecting to the database, be sure that you provide only limited information about the nature of the error to the user. If you disclose information related to data access and database errors, you could provide a malicious user with useful information that they can use to compromise your database security. Attackers use the information in detailed error messages to help deconstruct a SQL query that they are trying to inject with malicious code. A detailed error message may reveal valuable information, such as the connection string, SQL server name, table and database naming conventions, or the query being constructed.

### Tools and Libraries

* [Microsoft Source Code Analyzer for SQL Injection](http://www.microsoft.com/downloads/details.aspx?FamilyID=58a7c46e-a599-4fcb-9ab4-a4334146b6ba&displaylang=en). A static code analysis tool for finding SQL injection vulnerabilities in ASP code.
* [CAT.NET](http://www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en). The SDL also requires use of CAT.NET to perform XSS static analysis on ASP.NET applications. Download either version of CAT.NET depending on your needs—[32 bit](http://www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en) or [64 bit](http://www.microsoft.com/downloads/details.aspx?FamilyId=e0052bba-2d50-4214-b65b-37e5ef44f146&displaylang=en).

## Resources and Training for Developers

* [How To: Protect From SQL Injection in ASP.NET](http://msdn.microsoft.com/en-us/library/ms998271.aspx)
* [Preventing SQL Injections in ASP](http://msdn.microsoft.com/en-us/library/cc676512.aspx)
* [New SQL Truncation Attacks and How to Avoid Them](http://msdn.microsoft.com/en-us/magazine/cc163523.aspx)
* SQL Injection topic in [SQL Books Online](http://msdn.microsoft.com/en-us/library/ms161953.aspx)

# UNDERSTANDING SQL INJECTION VULNERABILITIES FOR THE TESTER/QA

To verify your code is protected from SQL injection attacks, you need to think like an attacker. This means you must be aware of how the attack is constructed and executed.

SQL injection vulnerabilities exist when a Web application accepts user input through HTTP requests, such as a GET or a POST, and then uses that input in the construction of a dynamic SQL statement. Here’s the simplest example:

1. The Web request looks like this:

GET http://www.somesite.com/book.asp?bookid=10

2. This Web request might return the details of the book based on the book id. As you can imagine, this page must have executed a **SELECT** statement based on the supplied bookid query string value.

3. Now, if the site is not validating the user input and constructs a dynamic SQL statement, an attacker could submit a Web request similar to the following:

http://www.somesite.com/book.asp?bookid=10;drop table books --

Even with this most simple of examples, there are numerous things an attacker could do with this link. As a tester, it is your job to think through what the attacker could do and attempt to successfully launch SQL injection attack against your solution, thus revealing a weakness to address in code.

To help ensure that you are executing effective tests, use the following processes to build your testing process before exercising your solution.

### ***Map Out the Site and Its Functionality***

Create some simple data flow diagrams that describe all the pages on the site and their purposes. Schedule meetings for threat modeling with the developers and architects/PMs, and use that time to drill into the application as much as possible. Gather the following data:

* Compile a complete list of public pages that return information from the database server without requiring user authentication.
* Make a list of all authenticated pages that execute SQL statements.
* Identify and document every point of user-supplied input.

Take the site map you create from this exercise a step further. Create a spreadsheet for this task. For each page, list out all of the query string parameters, cookie values, custom HTTP headers, POST data values, and other forms of user-supplied input passed. Don’t forget to search out Web services and similar SOAP requests, and identify all fields that allow user input. List every input parameter separately because you’re going to need to test each of them independently of the others.

### ***Start Testing and Pay Attention to the Output***

For SQL injection vulnerability, look for any deviation in the Web page response. Most of the time, if there is a SQL injection vulnerability, the database server returns a syntax error and the Web page might respond with an error page. Alternatively, you can start tracing the application using SQL Server profiler and look for parsing errors related to trace events from the Web application.

The same techniques can be applied to non-Web applications that execute SQL statements based on user input.

## Techniques for Finding Various Types of SQL Injection Vulnerabilities

### Code Reviews

If you are performing code reviews, here are some techniques for detecting issues in your SQL statements.

*Detecting First- or Second-Order SQL Injections*

* Identify the APIs used to execute SQL statements.
* Review whether any data validation is performed on the data used in the dynamic SQL statement.
* Confirm that the statements are executed using parameterized SQL where applicable (**SELECT**, **INSERT**, **UPDATE**, and **DELETE** statements).
* If data validation is not performed, review whether the data is escaped for delimiting characters (single quotes for string literals and right square brackets for SQL identifiers).

*Detecting SQL Modification by Truncation Issues*

* Review the buffer lengths used to store the final dynamic SQL statement.
* Calculate the maximum buffer required to hold the SQL statement when the input is maxed out and see if the buffer used to hold the SQL statement is big enough.
* Pay special attention to the return values from **QUOTENAME** or **REPLACE** functions. If the length of input data is *n* characters, these functions will return 2\*n + 2 or 2\*n when all the input characters are delimiting characters.
* For C/C++ applications, confirm that the return values from APIs like StringCchPrintf, which are used to prepare the SQL statement, are checked for insufficient buffer errors.

*Detecting SQL Injection by Truncation Issues*

* Review the buffer lengths that are used to store the delimited strings or escaped strings.
* If *n* is the length of the input string, then you need 2\*n + 2 for storing the return value from **QUOTENAME** and 2\*n for storing the return value from **REPLACE**.
* For C/C++ applications, confirm that the return values from replace equivalent functions are checked for insufficient buffer errors.

### Building Improvements into Black Box Testing[[9]](#footnote-9)

If you have automated tools or smart fuzzers, you can use the following techniques for detecting issues in your SQL statements.

*Detecting SQL Injection Issues*

* Send single quotes as the input data to catch the instances where the user input is not sanitized and used as string literals inside a dynamic SQL statement.
* Send double quotes as the input data to catch the instances where the user input is not sanitized and used as string literals inside a dynamic SQL statement.
* Use right square brackets (the ] character) as the input data to catch instances where the user input is used as part of a SQL identifier without any input sanitization.

*Detecting Truncation Issues*

* Send long strings, just as you would send strings to detect buffer overruns.

*Detecting SQL Modification by Truncation Issues*

* Send long strings of single quote characters (or right square brackets or double quotes). These max out the return values from **REPLACE** and **QUOTENAME** functions and might truncate the command variable used to hold the SQL statement.

## Tools You Can Use

Must-have tools for focused manual testing include:[[10]](#footnote-10)

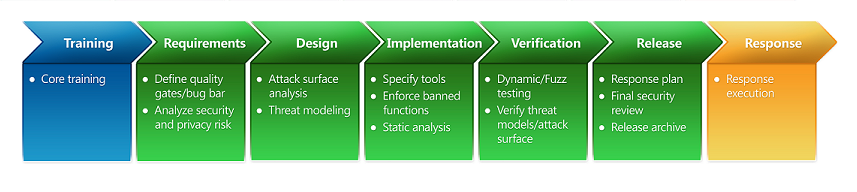
* [Fiddler](http://www.fiddlertool.com/fiddler) for intercepting HTTP traffic.
* [Paros Proxy](http://www.parosproxy.org/) for intercepting HTTP traffic.
* [Burp Proxy](http://www.portswigger.net/proxy/) for intercepting, inspecting, and modifying raw traffic in both directions.
* [TamperIE](http://www.bayden.com/dl/TamperIESetup.exe) for modifying GETs and POSTs.
* [HP Scrawlr](http://www.communities.hp.com/securitysoftware/blogs/spilabs/archive/2008/06/23/finding-sql-injection-with-scrawlr.aspx) for black box analysis of Web sites.

## Resources and Training for Testers

* [SDL Process Guidance, Verification Phase](http://msdn.microsoft.com/en-us/library/cc307418.aspx)
* [New SQL Truncation Attacks and How to Avoid Them](http://msdn.microsoft.com/en-us/magazine/cc163523.aspx)
* [Testing for SQL Injection](http://www.owasp.org/index.php/Testing_for_SQL_Injection)
* SQL Injection topic in [SQL Books Online](http://msdn.microsoft.com/en-us/library/ms161953.aspx)
* [*19 Deadly Sins of Software Security*](http://www.amazon.com/Deadly-Sins-Software-Security-Programming/dp/0072260858/ref=sr_1_1?ie=UTF8&s=books&qid=1256159028&sr=8-1) by Howard, LeBlanc, and Viega

# THE MICROSOFT SDL AND PREVENTING SQL INJECTION ATTACKS

SQL injection attacks can compromise entire enterprise systems and databases, steal users’ personal data, and distribute malicious content. SQL injection should be addressed as a serious threat to businesses and their customers. The Microsoft Security Development Lifecycle (SDL) addresses SQL injection by requiring database solutions, supporting applications, and online services teams to follow certain development standards throughout their development life cycle.

*Figure 4. Secure software development process model at Microsoft*

At a high level, the SDL details specific requirements for these types of solutions. Those requirements have been the core components of this paper. If these basic requirements are implemented as security requirements and adhered to throughout your software development process, you will have met the basic requirements related to SQL injection in the Microsoft SDL. These requirements include:

* Use parameterized SQL queries.
* Use stored procedures.
* Use SQL execute-only permission.
* Use proper escaping for SQL identifiers and Data Definition Language (DDL) statements.
* Handle buffer lengths properly.

However, the Microsoft SDL is not limited to the prevention of SQL injection attacks. It contains numerous policies and protections that help you write more secure code. Please visit the [SDL Web site](http://www.microsoft.com/sdl) to learn more about the SDL process, tools, partners, and training.

## Long-Term Solutions

The Microsoft SDL involves modifying an organization’s software development process by integrating measures that lead to improved software security and privacy. The intention of these modifications is not to totally overhaul the process but rather to add well-defined security checkpoints and security deliverables to help minimize the quantity and severity of security vulnerabilities in software.

By introducing security early in the software development life cycle, problems like SQL injection can be addressed in your underlying security requirements and discussed and mitigated in design. From a financial and business perspective, it is very beneficial to eliminate security problems as early as possible in the software development process. The National Institute of Standards and Technology (NIST) estimates that code fixes performed after software has been released are 30 times more costly to complete than fixes discovered and addressed during the design phase.

# CONCLUSION

SQL Injection is a major security problem plaguing the industry today. The ongoing mass SQL injection attacks reflect the prevalence of SQL injection vulnerabilities across thousands of Web sites. As SQL injection attacks continue to rise, it is imperative that development organizations prepare themselves with the solutions needed to rapidly address the problems as they occur. It is equally important that long-term solutions, including security policies/requirements, are in place to design, implement, verify, and release code that proactively protects your customers from SQL injection attacks.

# ACKNOWLEDGMENTS

Thanks to the following individuals for their contributions to this article:

Author: Bala Neerumalla

Contributors: Jeremy Dallman, Raul Garcia, David Ladd, Katie Moussouris, and Bryan Sullivan

1. Adapted from <http://www.verizonbusiness.com/resources/security/reports/2009_databreach_rp.pdf>. [↑](#footnote-ref-1)
2. Adapted from <http://www.breach.com/resources/whitepapers/downloads/WP_WebHackingIncidents_2008.pdf>. [↑](#footnote-ref-2)
3. Adapted from <http://www.pcworld.com/article/170457/getting_serious_about_sql_injection_and_the_tjx_hacker.html>. [↑](#footnote-ref-3)
4. Adapted from <http://news.cnet.com/Credit-card-breach-exposes-40-million-accounts/2100-1029_3-5751886.html>. [↑](#footnote-ref-4)
5. Adapted from <http://www.ftc.gov/os/caselist/0523148/0523148complaint.pdf>. [↑](#footnote-ref-5)
6. Adapted from <http://www.darkreading.com/security/perimeter/showArticle.jhtml?articleID=208804735>. [↑](#footnote-ref-6)
7. Adapted from <http://www.scmagazineus.com/Phishing-scams-may-target-Ameritrade-breach-victims/article/35699/?DCMP=EMC-SCUS_Newswire>. [↑](#footnote-ref-7)
8. Adapted from Michael Howard’s blog: <http://blogs.msdn.com/sdl/archive/2008/05/15/giving-sql-injection-the-respect-it-deserves.aspx>. [↑](#footnote-ref-8)
9. A adapted from <http://msdn.microsoft.com/en-us/magazine/cc163523.aspx>. [↑](#footnote-ref-9)
10. Adapted from <http://technet.microsoft.com/en-us/library/cc512662.aspx>. [↑](#footnote-ref-10)