# A Quick Introduction to SQL Injection

**Security Community Article – August 2009**
See other Security Community Articles

*by Brad Hill, Director of SDL Services and Geng Yang, Security Consultant; iSEC Partners*

SQL injection bugs are one of the favorite targets of Internet miscreants. These vulnerabilities can often be found and exploited in an automated fashion and may allow attackers extract entire databases, repurpose dynamic Web sites as malware distribution points, delete data and more. With automated attacks being launched against broad swaths of the Internet, every organization with a database connected to the Web needs to be concerned about preventing SQL injection. The good news is that these are vulnerabilities easy to fix – once they are found! This article will give you some tips and tricks to hunt down and eliminate SQL injection in your applications by:

- Preventing SQL injection by following best practices
- Finding vulnerabilities with code analysis tools
- Using black box tools for detection and defense
- Addressing SQL injection as a DBA

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SQL injection occurs when coding errors allow an attacker to modify the structure of SQL executed by a database server, changing the statement's logic to potentially malicious ends rather than those intended by the application. The most common cause is the use of unfiltered user input from a Web form in SQL statements constructed by string concatenation. Consider the following code snippet that builds a query to handle a user login:

```
var sqlText = "select * from users where user = '"

+ Request.form("user")

+ "' and password = '"

+ getSaltedHash(Request.form("password")) + "'";
```

This code is vulnerable because the user input from a Web form is directly inserted into the SQL text. Consider how specially-crafted user input like the following user name might affect the final structure of the statement:

```
Administrator'--
```

After concatenating the strings together, the SQL sent to the database server is:

```
select * from users where user = 'Administrator'--' and password = ''
```

Notice that the attacker has been able to change the structure of the query. They have used the apostrophe and double dash to prematurely terminate the query and comment out the remaining text, and can now log in as "Administrator", or any other user, without needing a password.

But this is not all they can do. The attacker might add a semicolon and insert additional commands to drop

tables, create new user accounts or use sub-selects to extract additional data. In fact, by using the computational capabilities of SQL it is often possible for the attacker to extract the entire structure and contents of a database, even if they can't see the results of a query displayed on the page. This technique is known as Blind SQL Injection, and a variety of tools exist to automate this process. Clearly, SQL injections are serious vulnerabilities. What can be done to identify, repair and ideally, prevent them?

---

## Preventing and Fixing SQL Injection Bugs

Avoiding SQL injection flaws is easy when a few best practices are followed.

**1 - Use parameterized queries**
Parameterized queries use placeholders for variable parameters, and bind the parameter to a specific data type before issuing the SQL statement. This method is effective because it ensures data is treated strictly as data, no matter what kind of statements are inside the data. Our sample login query above, built as a parameterized query would look like:

```
var sqlText = "select * from users where user = ? and password = ?"

Set param1 = objCommand.CreateParameter("user", adWChar, adParamInput, 50)

param1.value = Request.form("user")

objCommand.Parameters.Append param1

' and again for the password parameter...
```

**2 - Use stored procedures with static SQL**
A stored procedure is a subroutine available to applications accessing the SQL database. SQL injection cannot occur if only static SQL statements are used in stored procedures. If all data access logic is built into stored procedures, applications can be granted execute-only access to those procedures and disallowed from any other direct database access, severely limiting the potential for SQL injection.

Using stored procedures can eliminate application mistakes, but they must also be built securely. The following dynamic construction inside a stored procedure is vulnerable to SQL injection:

```
set @sql = 'select * from users where UserName = ''' + @username + ''''

exec sp_executesql @sql
```

Using a static, parameterized query removes the vulnerability:

```
ALTER PROCEDURE dbo.GetUserByUserName
(
@username varchar(20)
)
AS
SELECT * FROM User WHERE (UserName = @username)
RETURN
```

**3 - Avoid dynamic SQL whenever possible**

Parameterized queries are a best practice that should be preferred in all cases to building dynamic SQL, for both stored procedures and applications. Even "trusted" data (e.g. from a database instead of a Web form) may have been contaminated by malicious input at some time in the past. A SQL injection attack that enters from one part of a system and is triggered in another is known as a Second Order SQL Injection, and these are common in systems that rely on input filtering alone. Even if universally applied, input filtering that attempts to remove dangerous strings or escape single quotes can often be bypassed by a variety of techniques. Parameterized queries are the easiest and most sure way to use SQL securely.

In the few scenarios where parameterized statements cannot be used (for example, where column names are variable) input data must be carefully validated, preferably against a list of known good values. If arbitrary strings must be allowed, filter them to a very limited set of characters, such as alpha-numeric characters in the ASCII code page only, with no white space or punctuation.

**4 - Use LINQ**
The new Language Integrated Query (LINQ) technology available in Visual Studio "Orcas" and the .NET Framework 3.5 enables database constructs to be treated as native objects in .NET programming languages. LINQ to SQL abstracts an application's interactions with the database into an object model that avoids all possibility of SQL injection by automatically building parameterized queries.

**5 - Data validation**
Though these best practices should prevent SQL injection, all input should still be sanitized according to business logic rules to further reduce the possibilities for malicious content that could lead to second order SQL injection, stored cross-site scripting (XSS) or other vulnerabilities. Remember, too, that other query languages like LDAP or XPath can have vulnerabilities similar to SQL injection, which rigorous data validation will help prevent.

For existing applications that haven't followed these best practices everywhere, the problem remains of locating and repairing vulnerable code. Finding all the possible SQL injections in a large system can be difficult, but there are a variety of tools to assist in this process.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## Source Code Analysis Tools

For all but the smallest applications, using manual review to find code that is vulnerable to SQL injection is very time consuming. That's why Microsoft has released two free tools to help automate the process of identifying potential SQL injections. The first is the Microsoft Source Code Analyzer for SQL Injection, a command line tool to find vulnerabilities in Active Server Pages (ASP) code. This standalone, command-line tool can quickly scan ASP code written in VBScript and produces a list of warnings. For applications written in C# or Visual Basic .NET, the new Microsoft Code Analysis Tool for .NET (CAT.NET) is the tool of choice. CAT.NET performs advanced static analysis on the binary assemblies used by an application to find a variety of potential vulnerabilities, including, but not limited to, SQL injection. CAT.NET can be used as a standalone tool or as a snap-in for the Visual Studio IDE to display a list of links directly to source code where the flaw exists.

These tools (and other commercially available static analysis products) will greatly reduce the effort required to locate and fix SQL injections, but remember that no tool can be 100% accurate. For critical applications, rigid adherence to the development best practices outlined above is always the best defense.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## Black Box Detection and Defense

Sometimes it is necessary to test an application for which the source code is not readily available. A black box scanning tool can be of assistance here. While these tools are not able to find as many vulnerabilities as a source code scanner can, they have the advantage of being easily deployed by operational personnel and producing few false positives.

Scrawlr is one such tool. Developed by the HP Web Security Research Group in coordination with the MSRC specifically to respond to a series of widespread SQL injection attacks in 2008, Scrawlr is fast and free but does have some serious limitations. For example, it does not support sites that require authentication and does not test POST parameters, a major SQL injection attack surface. It can prove that code is badly broken, but cannot really demonstrate that code is safe or follows best practices. A variety of commercial Web application scanning products (including those made by HP) or on-demand, software-as-a-service offerings can provide much better coverage, but all black box scanners have their limitations. Source code informed analysis should be preferred or used in combination with these tools for the best results.

Assuming SQL injections have been found with one of these tools, what is the next step? Fixing the root cause of the vulnerability in the source code is the ultimate solution, but sometimes that cannot be done quickly. Black box filters, appliances and server plug-ins can provide a quick, temporary backstop until proper repairs can be made. For applications deployed on IIS, Microsoft provides the free URLScan tool which can be configured to eliminate many potential SQL injection payloads from HTTP headers and GET requests. Unfortunately, attacks carried in the POST body cannot be filtered by this tool. A Web Application Firewall (WAF) will offer a fuller feature set and the ability to filter all input to a vulnerable application. WAFs are available with a wide variety of features and deployment choices from both open source and commercial vendors. Filtering tools can be very useful, but always remember that they are stopgap measures, not substitutes for secure engineering practices and correct code.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## SQL Injections from the DBA's Perspective

It is common for database administrators to be responsible for SQL Server security. Is it possible to find SQL injections at the database tier? DBAs can and should review all stored procedures that call `EXECUTE`, `EXEC`, `sp_executesql` or any of the `xp_` extended functions to make sure they are preventing SQL injection. Weaknesses in the application tier cannot be directly identified at the database layer, but it is possible to get indirect evidence about whether developers are following best practices. Most applications connect to a database with a unique user account. Using the auditing functions of SQL Profiler for SQL Server, it is possible to record the SQL text of all the queries run by that user account. Counting the number of distinct query texts can give an idea of how that application uses SQL. A typical application using parameterized queries will have from a few dozen to perhaps a few hundred distinct SQL texts. Were the same application building dynamic statements from user data, there might instead be a unique SQL text for every query. While this technique cannot determine if an application follows best practices in all cases, any time an application is observed executing thousands or tens of thousands of unique queries, it is a good bet that that team could use some education about SQL injection.

A good way to prevent SQL injection at the database layer is to simply deny applications the ability to execute SQL and access the database directly. If all necessary application logic can be packaged into properly-audited stored procedures, the application domain group can be granted execute-only permission on those stored procedures, and no other database permissions. This strategy requires additional planning but can provide a

very strong defense.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

## Conclusion

SQL injection vulnerabilities are one of the most serious classes of application security flaw. Even a single piece of vulnerable code can let an attacker access important data or control an application. For mission-critical systems, the techniques discussed here should be employed in combination for the strongest security assurance. At iSEC Partners, we recommend our clients prevent and eliminate SQL injection by employing a proven and comprehensive approach, like the Microsoft Security Development Lifecycle (SDL), that builds security in at every phase of the software lifecycle:

- Training – Educate developers about SQL injection and data access best practices.
- Design – Design with secure technologies such as LINQ and stored procedures.
- Implementation – Use static analysis tools to locate and eliminate vulnerable code.
- Verification – Use black box tools as part of quality assurance testing.
- Response – Monitor and prepare for rapid mitigation of vulnerabilities missed by other processes.

Be sure to visit MSDN online for much more information on SQL injection, the technologies discussed in this article, and the Microsoft Security Development Lifecycle (SDL).

---

*Brad Hill is the Director of SDL Services and Geng Yang is a Security Consultant at iSEC Partners, a full-service security consulting firm that provides penetration testing, secure systems development, security education and software design verification. Visit* https://www.isecpartners.com *for more information.*