



octavient

SQL Injection Defense Methodology

Author: Wayne Saucier

Date: 07/25/2006

Version: 1.0

Non-Technical Executive Summary

SQL Injection is a well-known infiltration technique inherent to public-facing web applications (or mere web sites) that access data stored in a database (SQL is the language used to access data within most modern database systems). It's a commonly overlooked security vulnerability, because it is exploited using the very same web pages and scripts used by legitimate users. Instead of looking for a back-door to get into your database or your network, SQL Injection exploiters use the front door—simply by entering malicious syntax into web page form fields.

For example, a common exploit involves hijacking a common login page with a username and password field. With no protection, the database can be exploited by submitting this form with a blank password and the characters [`'OR 1=1;--`] (without the brackets) in the username field.

This syntax is designed to hijack the script that's hard-coded within the program and that is designed to perform Function A (checking the list of stored usernames and passwords) and cut it off and instead perform Function B (checking whether the statement "1=1" is true, which it is).

So, instead of saying:

```
Check the database where
Username='<User Input>' AND Password='<User Input>' ;
```

which grants the user access to the system when the username and password match with a combination stored in the database, the script now says:

```
Check the database where
Username='' OR 1=1 ;
```

which is always true, granting access to the system to anybody smart enough to try that hack.

This example is merely one of hundreds of SQL statements that can be injected. With the various permutations introduced by using encoded characters (`&12;=&12;`), whitespace (`12 = 12`), and/or logical permutations (`2=2` or `'firetruck'='firetruck'`), the vulnerability variations are nearly limitless.

The vulnerability also exposes more than just the web application or the database themselves—the technique can be combined with privilege escalation techniques to infiltrate the internal network in which the database resides.

While this vulnerability is no less complex than any other type of network security vulnerability, the baseline techniques for defense are simple enough, and are usually enough to send potential infiltrators to pursue lower-hanging fruit elsewhere (if you're not a lucrative target, such as a bank or government institution).



Vulnerability Overview

SQL Injection Vulnerabilities exist in web applications where malicious users can inject SQL syntax into page requests that will perform unintended functions against the database, such as fooling form-space authentication barriers, or displaying RowSets, or even Creating or Dropping tables.

Such a vulnerability exists at every point in an application where SQL statements are executed against a database (of any platform) using parameters input by an end-user. This can include not only log-in pages, but any other interactive page within an application, whether they're designed to be used by registered (paying or not) or anonymous users.

This can also include dynamic web pages that only pull data from a database.

The vulnerability is also not limited to standard HTML forms—user input can also be supplied via query strings, client-side cookies, and browser environment values such as user agent strings and IP addresses.

The vulnerability can also extend beyond the application or the database themselves; a SQL injection attack could potentially allow an intruder to create tables with binary fields, upload penetration-enabling executables, and gain access to a server's internal network with privilege escalation.

General Defense Principles

- Assume all user input is malicious.
- Remember that no data or programming platform is automatically immune.
- Use least-privileged account for data access.
- Validate and scrub all end-user input (with server-side validation; NOT client-side).
- No single defense method is impermeable; employ as many methods as possible.
- Use SQL queries that are as controlled and parameterized as possible.
- Sensitive data should never be stored in plain-text.
- Obscure all error messages.
- Employ a change control methodology that ensures new vulnerabilities aren't created as an application is expanded.
- Vulnerabilities change over time; a routine testing regimen should be employed, and should accommodate recognition of evolving penetration techniques (such as with an actively-managed set of SQL Injection signatures). Because this type of vulnerability doesn't evolve as quickly as other known IT vulnerabilities, the schedule need not be as aggressive as general network penetration testing or antivirus signature updating, for example.

General Defense Methodologies

- Use a limited access account for access to the database created specifically for the application (avoid use of the SA account).
- Create a read-only access account for the login-page (where the user only needs to read the database).



- Use encryption on db-stored passwords, credit card numbers, etc. (this prevents the type of attack that displays all records from a table)
- Use parameterized SQL queries or stored procedures for data access.
- Limit the error info that is returned to the user (i.e., use validation to return custom error messages that don't offer detail that could be used to determine the level of vulnerability).
- Validate **ALL** data input by end users that will touch the database.
 - **Constrain:** Check for known good data by validating the type, length, format, and range. Use validation controls native to the programming platform employed and/or regular expressions and custom validation.
 - **Reject:** Compare data input to known malicious syntax (including the various character encoding methods (HTML, UTF, Unicode, ISO, etc.) stored as a set of signatures. Reject known malicious input with obscure, custom error messages.
 - **Use intelligent validation.** Known malicious syntax can be easily obscured, with whitespace, character encoding, etc.
 - **Sanitize:** Data input should be TRIMmed, and sanitized (i.e., encode characters commonly used in malicious input, such as the single-quote, the double-hyphen or the semi-colon).

Testing Methodologies

- Employ a reputable automated vulnerability scanner, such as Nessus.
- Testing should be conducted against every script that dynamically creates (and executes) a SQL statement with user input.
- Change control should incorporate testing new scripts and pages that are created when an application is expanded.
- Employ a custom testing regimen, including these types of tests:
 - Attempt to inject dangerous characters (or text string) using query string parameters.
 - Attempt to inject encoded dangerous characters.
 - Use a known sanitization of a dangerous character at the end of a max-field input.
 - Use safe characters in a way that will create SQL error messages (to test smart-error-message-prevention techniques).