SQL Injection protection in the DataFlex Environment

A compilation by Marcia Booth – Data Access Worldwide
September 2, 2015

Overview

This document provides information about SQL Injection attacks and how the risks associated with them are mitigated in the DataFlex application environment. When SQL statements written by a developer are included in an application program for direct execution by an SQL server, the developer is responsible for making sure that his or her programming style does not let the guard down for threats like SQL Injection or other risks.

What is an SQL Injection attack?

"SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives. Even parameterized data can be manipulated by a skilled and determined attacker." Source: MICROSOFT

How can SQL Injection affect my DataFlex application?

DataFlex applications using SQL are probably using one of Data Access' DataFlex Connectivity Kits which can use SQL in two ways:

As converted DataFlex database requests

DataFlex database requests (find, save, etc.) are controlled by the compiled DataFlex program and must be converted to SQL requests. For each database request, the DataFlex Connectivity Kit creates SQL statements with parameters, also known as prepared statements. That means that those DataFlex generated SQL statements are compiled on the server, then user input is assigned to the prepared statement as parameters making it much, much harder—almost impossible – to implement SQL injection attacks from input to a standard DataFlex program.

"The reason that prepared statements help so much in preventing SQL injection is because the values that will be inserted into an SQL query are sent to the SQL server after the actual query is sent to the server. In other words, the data input is sent separately from the prepared query statement. This means that there is absolutely no way that the data input can be interpreted as SQL, and there's no way that a hacker could run his own SQL on your application. Any input that comes in is only interpreted as data, and cannot be interpreted as part of your own application's SQL code." Source: PROGRAMMERINTERVIEW.COM

For example, if the structure of an update instruction for a row in a table is converted to

```
UPDATE table SET field1 = ?, field2 = ?, … WHERE RECNUM = ?
```

by the DataFlex Connectivity Kit, the statement will then be compiled via an SQLPrepare operation, then, if successful, the user input data is added to the compiled statement with SQLBind operations and finally an SQLExecute is performed to complete the update and modify data in the table.

So for table operations, since the combination of a DataFlex program and Connectivity Kit uses parametrized queries, i.e. prepared execution, applications are much safer from SQL injection.

As direct statements to be executed by your application

Direct statements are SQL statements embedded by a developer in an application's program code. In such a case, the SQL statements are executed as written by the developer assuming they are valid.

Since direct SQL statements are being written by the developer and not an end user, direct statements can be safe if written competently and with knowledge of SQL Injection attack avoidance. However, if SQL code is written in a way that makes it vulnerable to SQL injection, the application will be then vulnerable, too. In other words, when using embedded SQL, an application is as safe as the direct statements that the developer writes. The developer must take responsibility for his own code; DataFlex cannot protect against developers' risky SQL code.

The only place that DataFlex itself uses SQL directly is through the Data Dictionary (DD) SQL filters and those filters are set by the developer. To protect against SQL Injection attacks, DataFlex DD SQL filters have their strings double checked by "escaping them" (see below) on the small chance that the filter strings are obtained from an end-user via some kind of user input. In DataFlex for Linux there are no DD SQL filters so the above does not apply.

What can be done to prevent SQL Injection Attacks?

When using embedded SQL and DD SQL filters, developers should avoid allowing user input to be used directly in SQL statements. If, for example, a program has a 'search' input where a user can compose any text (including SQL code!) as a search string, that text should not be used directly to compose an SQL statement. The developer should make sure that any such user input is first escaped and validated, and then used.

To assist developers in this task, DataFlex includes a function called **SQLEscapedStr** that changes a string to an "escaped" string that is better suited for being used in direct SQL statements. This function replaces a single quote with two single quotes, which helps protect against SQL injection.

In the programming world, *escaping* means "allowing special characters (like single/double quotes, percent signs, backslashes, etc.) in strings to be saved so that they remain as part of the string, and are not misinterpreted as something else. For example, if we want to include a single quote in a string (like in the string "it's") that gets output to the browser […], then we have to add a backslash to the single quote so that it is still interpreted as a single quote when generating the output." Source: PROGRAMMERINTERVIEW.COM

Also, developers writing embedded SQL should always validate user input by testing data type, length, format, and range. When implementing precautions against malicious input, developers should consider the architecture and deployment scenarios of the application. Remember that programs designed to run in a secure environment can be copied to a non-secure environment. The following suggestions should be considered best practices for embedded SQL:

- Make no assumptions about the size, type, or content of the data that is received by your application. For example, developers should make the following evaluations:
    - How will your application behave if an errant or malicious user enters a 10-megabyte MPEG file where your application expects a postal code?
    - How will your application behave if a DROP TABLE statement is embedded in a text field?
- Test the size and data type of input and enforce appropriate limits. This can help prevent deliberate buffer overruns.
- Test the content of string variables and accept only expected values. Reject entries that contain binary data, escape sequences, and comment characters. This can help prevent script injection and can protect against some buffer overrun exploits.
- When working with XML documents, validate all data against its schema as it is entered.
- Never build Transact-SQL statements directly from user input.
- Use stored procedures to validate user input.
- In multitier environments, all data should be validated before admission to the trusted zone. Data that does not pass the validation process should be rejected and an error should be returned to the previous tier.
- Implement multiple layers of validation. Precautions you take against casually malicious users may be ineffective against determined attackers. A better practice is to validate input in the user interface and at all subsequent points where it crosses a trust boundary.

  For example, data validation in a client-side application can prevent simple script injection. However, if the next tier assumes that its input has already been validated, any malicious user who can bypass a client can have unrestricted access to a system.

- Never concatenate user input that is not validated. String concatenation is the primary point of entry for script injection.

- Do not accept the following strings in fields from which file names can be constructed: AUX, CLOCK$, COM1 through COM8, CON, CONFIG$, LPT1 through LPT8, NUL, and PRN.

Whenever possible, input that contains the following characters should be rejected:

Input character	**Meaning in Transact-SQL**
;	Query delimiter.
'	Character data string delimiter.
--	Comment delimiter.
/* ... */	Comment delimiters. Text between /* and */ is not evaluated by the server.
xp_	Used at the start of the name of catalog-extended stored procedures, such as **xp_cmdshell**.

Source: MICROSOFT TECHNET

For further reading on best practices in validating user input, access SQL INJECTION.

Sources

John Tuohy, CTO, Data Access Worldwide

Martin Moleman, Software Engineer, Data Access Europe

DataFlex and SQL Injection

http://support.dataaccess.com/Forums/entry.php?109-DataFlex-and-SQL-Injection

How to prevent SQL injection attacks?

http://www.programmerinterview.com/index.php/database-sql/sql-injection-prevention/

SQL Injection

https://technet.microsoft.com/en-US/library/ms161953(v=SQL.105).aspx