

## Beyond Stored Procedures: Defense-in-Depth against SQL Injection

A few years ago, mentioning the phrase “SQL Injection” to developers or asking to adopt a “defense-in-depth” strategy would probably get you a blank stare for a reply. These days, more people have heard of SQL Injection attacks and are aware of the potential danger these attacks present, but most developers’ knowledge of how to prevent SQL Injection is still inadequate, and when asked how to defend their applications against SQL Injection, they usually reply, “That’s easy, just use stored procedures.” As we will see, using stored procedures is a great first step for your defense strategy, but is not sufficient as the only step. You need to adopt a defense-in-depth strategy.

If you are not familiar with SQL Injection attacks and their potential for danger to your applications, please see the MSDN article “SQL Injection” (<http://msdn2.microsoft.com/en-us/library/ms161953.aspx>).

The problem with exclusively relying on stored procedures and not implementing a defense-in-depth strategy is that you are really just counting on the developer of the stored procedures to provide your security for you. Stored procedures, similar to the following SQL Server code used to authenticate a user, are fairly common:

```
ALTER PROCEDURE LoginUser
(
    @UserID [nvarchar](12),
    @Password [nvarchar](12)
)
AS
SELECT * FROM Users WHERE UserID = @UserID AND Password = @Password
RETURN
```

That stored procedure looks pretty secure, but consider this one:

```
ALTER PROCEDURE LoginUser
(
    @UserID [nvarchar](12),
    @Password [nvarchar](12)
)
AS
EXECUTE ('SELECT * FROM Users WHERE UserID = ''' + @UserID + ''' AND Password
= ''' + @Password + ''')
RETURN
```

By creating an ad-hoc SQL statement and passing it to the EXECUTE function in the code of stored procedures, we can actually create SQL-injectable stored procedures. This is even easier to do when you use managed code to write stored procedures, as is newly supported in Microsoft SQL Server 2005:

```
[Microsoft.SqlServer.Server.SqlProcedure]
public static void LoginUser(SqlString userId, SqlString password)
{
    using (SqlConnection conn = new SqlConnection("context connection=true"))
    {
```

```

        SqlCommand selectUserCommand = new SqlCommand();
        selectUserCommand.CommandText = "SELECT * FROM Users WHERE
UserID = '" + userId.Value + "' AND Password = '" + password.Value + "'";
        selectUserCommand.Connection = conn;

        conn.Open();
        SqlDataReader reader = selectUserCommand.ExecuteReader();
        SqlContext.Pipe.Send(reader);
        reader.Close();
        conn.Close();
    }
}

```

Even if you are the one writing the stored procedures, you usually cannot be sure that someone else will not come behind you and change them after the application has been deployed. This is especially true regarding Web applications and is why a defense-in-depth strategy can help.

Clearly, the solution to the problem is to adopt a defense-in-depth strategy. You should continue to use stored procedures and parameterized queries whenever possible, but you should also take steps in establishing your defense-in-depth strategy to ensure that the parameters passed to those stored procedures and queries are validated. In our user authentication example above, "bobsmith" may be a valid user ID, but "SELECT \* FROM tblCreditCards" is probably not. A good way to use your defense-in-depth strategy to validate user input is to apply regular expression rules to it. You can use the `RegularExpressionValidator` control found in the `System.Web.UI.WebControls` namespace to validate Web form data, and you can use the `Regex` class found in the `System.Text.RegularExpressions` namespace to validate any kind of text data. Here is an example of a Web form that validates the user input before passing it off to the database:

```

protected void Page_Load(object sender, EventArgs e)
{
    if (Page.IsPostBack)
    {
        // We allow only alphanumeric input
        Regex allowRegex = new Regex("^[a-zA-Z0-9]*$");
        if ((!allowRegex.IsMatch(textBoxUserId.Text)) ||
(!allowRegex.IsMatch(textBoxPassword.Text)))
        {
            labelErrorMessage.Text = "Invalid user ID or password.";
            return;
        }
        else
        {
            // Call the login stored procedure
            ...
        }
    }
}

```

An even more thorough defense-in-depth strategy is to use a combination of allowed-input patterns (also known as 'whitelists') and denied-input patterns (or 'blacklists'). The user's input must match the whitelist pattern (or at least one of the whitelist patterns, if there is more than one) and not match the blacklist pattern (or any of the blacklist patterns). You should definitely consider using blacklist patterns

with your defense-in-depth strategy if you allow non-alphanumeric input such as apostrophes in your allowed-input list.

```
// We allow alpha characters, spaces, and apostrophes as input
Regex allowRegex = new Regex(@"^[a-zA-Z\s\']*");
// But we disallow common SQL functions
Regex disallowRegex = new Regex("(union|select|drop|delete)");
if ((!allowRegex.IsMatch(textBoxLastName.Text)) ||
    (disallowRegex.IsMatch(textBoxLastName.Text)))
{
    labelErrorMessage.Text = "Invalid name.";
    return;
}
```

Finally, we have to address the question of adding damage control to your defense-in-depth strategy. If a hacker did find a way to execute SQL commands against your database, what kind of damage could he/she do? If your application connects to the database as an administrative user, such as "sa" for Microsoft SQL Server, the damage could be severe indeed. Not only could he/she view the data in the tables, he/she could add new data of his own, or change the values of the existing data. Imagine an online shopping site where all items have had their price marked down to a penny. He/she could add new users or remove existing users. He/she could delete rows, tables, or even the entire database. You can alleviate this risk by applying the principle of least privilege to your defense-in-depth strategy: make your application connect to the database as a user who has just enough permissions to perform the actions required, and no more. If your application only reads data from a database, remove the insert, update, and delete permissions for the database user. If the application only needs access to a product catalog database (for example), make sure the user has no access to the order history database. Never specify "sa" or any administrative user as the database user.

By adopting a defense-in-depth strategy, you can avoid most or all of the damage that a SQL Injection attack can cause to your application. It is a great idea to use stored procedures for many reasons, including improved security, but do not rely on them to provide all of your security. Always validate user input, and apply the principle of least privilege to minimize the damage that a successful attack can cause.

### **About the Author**

Bryan Sullivan is a development manager at SPI Dynamics, a [Web application security products company](#). Bryan manages the DevInspect and QAInspect Web security products, which help programmers maintain [application security](#) throughout the development and testing process. He has a bachelor's degree in mathematics from Georgia Tech and 11 years of experience in the information technology industry. He also contributed to the AVDL specification, which has become a standard in the application security industry.