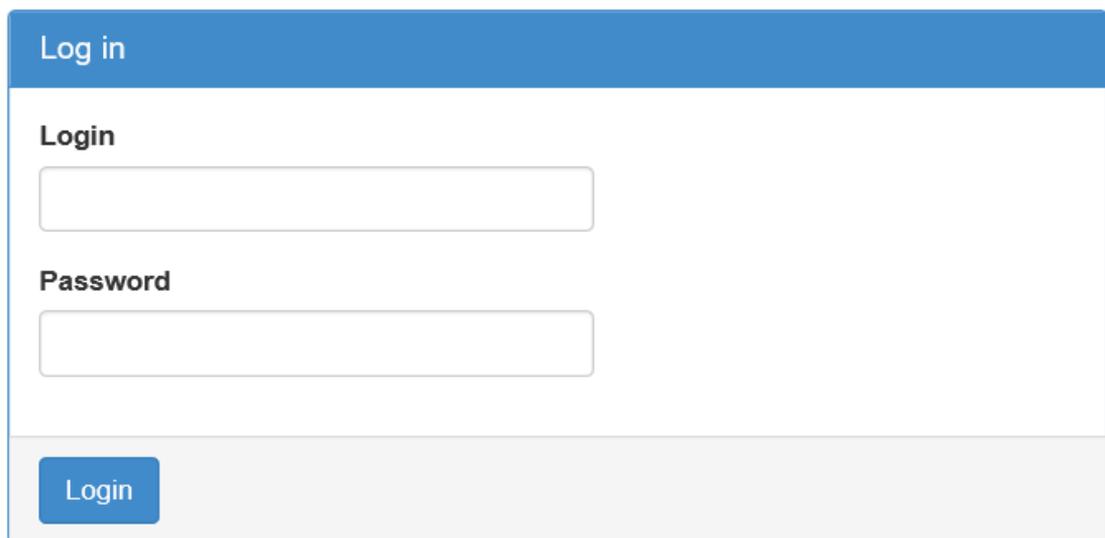


Avoid SQL Injection Attacks When Using Dynamic SQL

While there are many programmers who use the Entity Framework these days for database access, there are still many who do not. Many programmers still use ADO.NET directly to perform standard CRUD logic. When you must use dynamic SQL to perform queries, it is very important to not let any SQL injection attacks through.

A SQL injection is a technique whereby a hacker will attempt to damage your data by submitting SQL through one of your web pages. If you are using dynamic SQL to build a SQL statement that will be submitted to your database, make sure you are doing it correctly. To illustrate a SQL injection attack, let's look at a typical login page like that shown below:



The image shows a login form with a blue header bar containing the text "Log in". Below the header, there are two input fields: one labeled "Login" and one labeled "Password". At the bottom of the form, there is a blue button labeled "Login".

When the user clicks on the Login button, you are going to look in two columns of your AppUser table to see if the data input matches the data in any of the rows of this table. Below is a script that will create the AppUser table and add some data.

```
CREATE TABLE AppUser
(
    Login char(16) not null primary key,
    Password varchar(20) not null
);

INSERT INTO AppUser values('PaulS', 'password');
INSERT INTO AppUser values('JohnK', 'password');
```

You can copy and paste the above SQL code into your database management system and create this table. To create the login screen, build a LoginViewModel class with two properties to bind to the two fields on the web page.

```
public class LoginViewModel
{
    public string Login { get; set; }
    public string Password { get; set; }
}
```

After you have created the table and the LoginViewModel class, build the login page using Bootstrap classes.

```
@model DynamicSQLSample.LoginViewModel

@{
    ViewBag.Title = "Login Page";
}

@using (Html.BeginForm()) {
    <div class="row">
        <div class="col-xs-12 col-sm-6">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3 class="panel-title">Log in</h3>
                </div>
                <div class="panel-body">
                    <div class="form-group">
                        @Html.LabelFor(m => m.Login)
                        @Html.TextBoxFor(m => m.Login,
                            new { @class = "form-control" })
                    </div>
                    <div class="form-group">
                        @Html.LabelFor(m => m.Password)
                        @Html.TextBoxFor(m => m.Password,
                            new { @class = "form-control",
                                type = "password" })
                    </div>
                </div>
                <div class="panel-footer">
                    <button type="submit" class="btn btn-primary">
                        Login
                    </button>
                </div>
            </div>
        </div>
    </div>
</div>
}
```

Add Login Method

Now that you have the page created and the view model, add a Login() method to the LoginViewModel class.

```
public bool Login()
{
    bool ret = false;
    string sql = string.Empty;
    string conn = string.Empty;
    int rows = 0;
    SqlCommand cmd = null;

    conn = @"Server=(LocalDB)\MSSQLLocalDB;";
    conn +=
@"AttachDbFilename=|DataDirectory|\LoginSample.mdf;";
    conn += "Integrated Security=True;";

    sql = "SELECT COUNT(*) As TotalRows FROM AppUser ";
    sql += "WHERE Login = '{0}' AND Password = '{1}'";
    sql = string.Format(sql, this.Login, this.Password);

    using (cmd = new SqlCommand(sql, new SqlConnection(conn))) {
        cmd.Connection.Open();

        rows = Convert.ToInt32(cmd.ExecuteScalar());

        ret = rows > 0;
    }

    return ret;
}
```

Create the Controller

In the controller for your login page, write the Get and Post methods.

```
public ActionResult Index()
{
    LoginViewModel vm = new LoginViewModel();

    return View(vm);
}

[HttpPost]
public ActionResult Index(LoginViewModel vm)
{
    vm.Login();

    // TODO: Do something after attempting login

    return View(vm);
}
```

Set a breakpoint in the Login() method, run the page and enter a valid login and password into the fields and click on the Login button. If you wrote the code correctly, you should see one row returned from the ExecuteScalar() method. While the above method works, let's learn how a hacker would take advantage of this web page to potentially damage your data. This does assume that the hacker has found out the name of the table you are using to store your data. But, if they find out, then they would enter the following text into your login field. They can just type any text into the password field.

```
' ; DELETE FROM AppUser; --
```

The above string entered into your login field will create SQL that looks like the following:

```
SELECT Count(*) As TotalRows
FROM AppUser WHERE Login = '';

DELETE FROM AppUser; --' AND Password = ''
```

The first single quote entered by the hacker closes the beginning quote of the Login field. This makes the WHERE clause valid. The SQL parser will then consider this a valid SQL statement and execute it. The next statement is DELETE From AppUser; followed by two dashes. This tells the SQL parser to ignore the rest of the text. The net effect is you get zero rows returned, but all rows in the AppUser table have been deleted!

Use Parameters

With just a few minor changes to the code in the Login() method, you can protect yourself against this type of attack. Instead of using string.Format() or otherwise just concatenating your SQL and user input together, use SqlParameter objects.

First off, modify the lines of code that build your SQL string, to look like the following:

```
sql = "SELECT COUNT(*) As TotalRows FROM AppUser ";  
sql += "WHERE Login = @Login AND Password = @Password";
```

Next, after the using() statement and before you open your connection, add the following lines of code to build parameters on the SqlCommand object.

```
using (cmd = new SqlCommand(sql, new SqlConnection(conn))) {  
    cmd.Parameters.Add(  
        new SqlParameter("@Login", this.Login));  
    cmd.Parameters.Add(  
        new SqlParameter("@Password", this.Password));  
  
    cmd.Connection.Open();  
}
```

All of the rest of the code in the Login() method stays the same. When you use SqlParameter objects, the ADO.NET SQL engine ensures no SQL injection attacks can occur. This is much better than you trying to check all input to make sure there is no potentially harmful code. The Entity Framework uses SqlParameter objects to submit all SQL to the database.

Summary

If you still use ADO.NET to submit SQL to your database, make sure you are using SqlParameter objects and are not passing user data directly to your database. With just a few extra lines of code, and changing how you create your SQL statements you ensure you will not be a victim of a SQL injection attack.