

Building a Defensive Perimeter Against Malicious User Input

by Jeff Prosise

Quick: Can you spot two fatal flaws in the following ASP.NET Web page?

```
<html>
  <body>
    <form runat="server">
      <asp:TextBox ID="Input" RunAt="server"
/>
      <asp:Button Text="Click Me"
OnClick="OnSubmit" RunAt="server" />
      <asp:Label ID="Output" RunAt="server" />
    </form>
  </body>
</html>

<script language="C#" runat="server">
void OnSubmit (Object sender, EventArgs e)
{
    Output.Text = "Hello, " + Input.Text;
}
</script>
```

Don't bother looking for syntax errors: functionally, this page is just fine. Type your name into the input field and click the "Click Me" button, and a personalized greeting appears on the page. But security-wise, this page is a disaster-despite the fact that it's similar to countless sample pages found in books, magazine articles, and even the .NET Framework SDK.

Give yourself a pat on the back if you spotted the following flaws:

- The page neither constrains nor validates user input
- It echoes raw, unfiltered user input to the page

Constraining and validating user input is absolutely essential in a Web application to prevent hack attacks that rely on malicious input strings. SQL injection attacks are one example of such hacks. Suppose the page were modified to look like this:

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<html>
  <body>
    <form runat="server">
      <asp:TextBox ID="Input" RunAt="server" />
      <asp:Button Text="Click Me" OnClick="OnSubmit"
RunAt="server" />
      <asp:DataGrid ID="Output" RunAt="server" />
    </form>
  </body>
</html>

<script language="C#" runat="server">
void OnSubmit (Object sender, EventArgs e)
{
    SqlDataAdapter adapter = new SqlDataAdapter
        ("select * from customers where country='" +
Input.Text + "'",
        "server=localhost;database=northwind;uid=sa");
    DataSet ds = new DataSet ();
    adapter.Fill (ds);
    Output.DataSource = ds;
    Output.DataBind ();
}
</script>
```

The modified page queries SQL Server's Northwind database and displays the records in the Customers table that include the country name typed into the input field. Type "USA" and you'll see a list of Northwind customers in the United States; type "UK" and you'll see customers in the United Kingdom.

SQL injection is a very real threat against this page because `OnSubmit` plugs the country name that the user types into a SQL `SELECT` statement and executes it. Want to delete part of the Northwind database? Try typing this into the input field and clicking the button:

A Word About Secure Database Access

Unfortunately, this database access code exhibits insecurities of its own. For example, you should never use the `sa` account (or an equivalent) to access databases from Web applications. Instead, use weak accounts that lack permission to drop tables, insert, update, and delete records, and the like. In addition, you should use stored procedures or parameterized commands in lieu of dynamic SQL commands for added protection against malicious input parameters. Finally, consider encrypting database connection strings to minimize the risk of information disclosure if your source code falls into the wrong hands. And note that truly paranoid ASP.NET programmers encrypt connection strings and store them in ACLed registry keys. When it comes to Web security, a little paranoia can be a good thing.

```
usa';drop table orders--
```

The developer who built this page probably envisioned queries like this one being performed:

```
select * from customers where country='usa'
```

But the malicious input string fundamentally changes the query to include a DROP TABLE command:

```
select * from customers where  
country='usa';drop table orders--'
```

Because -- is SQL's comment operator, everything following the DROP TABLE command is ignored. Because the attacker was *allowed* to enter malicious input, the database on the back end is exposed. And that's just one of many ways to perpetrate SQL injection attacks against this page.

Echoing raw, unfiltered user input to a Web page is equally dangerous. To demonstrate, try typing the following text into the input field using the original version of the page:

When you click the "Click Me" button, a message box appears. That in and of itself isn't harmful, but it tells hackers that the page is vulnerable to cross-site scripting attacks. Cross-site scripting can be used to steal cookies, deface Web sites, and, in extreme cases, breach firewalls and gain access to remote databases.

Here are two simple rules for preventing user input from being used as a carrier for attacks against your Web site:

- Religiously constrain and validate input, whether it comes directly from the user (for example, from a text input field) or elsewhere (for example, from a query string)
- Always HTML-encode user input before outputting it to a Web page

Input can be constrained and validated in a number of ways. ASP.NET's `RegularExpressionValidator` control, for example, is a fine tool for restricting what users can enter from text fields and filtering out potentially malicious characters such as single quotes, angle brackets, and hyphens. The `Regex` class located in the .NET Framework Class Library's `System.Text.RegularExpressions` namespace is equally adept at validating query strings. The `TextBox` control's `MaxLength` property is also a valuable tool in the war on hackerism because more sophisticated SQL injection attacks require commensurately longer input strings. However you go about validating input, don't make the mistake of trying to filter out certain characters or character sequences while allowing everything else; hackers will use various character encodings in attempts to bypass your filters. The better approach is to define what you *will* accept (for example, letters and numbers in user names) and reject everything else.

HTML-encoding input before using it as output is important because it converts certain potentially dangerous characters such as angle brackets into harmless HTML. Take this text string, for example:

```
<script> alert('Gotcha!') </script>
```

ASP.NET 1.1 Request Validation

The message box appears just fine if you're running ASP.NET 1.0, but it won't appear at all on ASP.NET 1.1 unless you add the following statement to the ASPX file:

```
<%@ Page  
validateRequest="false" %>
```

By default, ASP.NET 1.1 automatically inspects HTTP requests and rejects input that it deems dangerous. While useful, the reality is that many companies turn this feature off because they find it too restrictive. (It's impossible to build a generic filter that satisfies the needs of everyone.) Therefore, the onus remains on the developer to defend against cross-site scripting attacks. Even if your company leaves ASP.NET's request validation feature on, do you want the security of your code to depend on a configuration setting?

If written without modification to a Web page, it looks to a browser to be legitimate client-side script and is therefore executed. But if the same string is HTML-encoded, it looks like this:

```
<script>alert('Gotcha!')</script>
```

Now the script doesn't get executed because the browser no longer views it as script. Furthermore, the string is now accurately displayed on the page it's echoed to, whereas before-when it was interpreted as client-side script-it wasn't displayed at all.

Below is a revised version of the page that incorporates these fixes, with changes highlighted in bold. A `MaxLength` attribute limits input strings to 16 characters, while a `RegularExpressionValidator` allows through only letters of the alphabet. Furthermore, input strings are HTML-encoded with `HttpUtility.HtmlEncode` before being written to the page. This page is substantially more secure than the original one. It's unlikely that an attacker could find a way to exploit it even if it did access a back-end database.

```
<html>
<body>
  <form runat="server">
    <asp:TextBox ID="Input" MaxLength="16"
    RunAt="server" />
    <asp:Button Text="Click Me" OnClick="OnSubmit"
    RunAt="server" />
    <asp:RegularExpressionValidator RunAt="server"
    ControlToValidate="Input"
    ValidationExpression="[a-zA-Z]*"
    ErrorMessage="Invalid input" Display="dynamic" />
    <asp:Label ID="Output" RunAt="server" />
  </form>
</body>
</html>

<script language="C#" runat="server">
void OnSubmit (Object sender, EventArgs e)
{
  Output.Text = "Hello, " + HttpUtility.HtmlEncode
  (Input.Text);
}
</script>>
```



The number one rule for writing secure Web code is to assume that input is malicious and code accordingly. That means validating all input and HTML-encoding it before echoing it to a page. You can never be absolutely certain that a site isn't vulnerable to attacks, but you can make sure that an attacker will have to work very hard to breach your defenses.

Jeff Proise is a Wintellect cofounder, an expert in ASP.NET, and a passionate believer that the world would be a better place if hackers were publicly flogged. He also teaches beginning, intermediate, and advanced ASP.NET programming classes. Click [here](#) for more information.

Normalizing Requests Using ISO-8859-1 Encoding

Even with `RegularExpressionValidators` standing sentinel over input, hackers will attempt to get past them by using alternate request encodings to slip malicious characters past input filters. As an added precaution, add the following statement to `Web.config` to "normalize" requests using ISO-8859-1 encoding:

```
<globalization
requestEncoding="ISO-8859-1"
responseEncoding="ISO-8859-1"
/>
```

This raises the bar even higher for attackers, further reducing the chance that a malicious character will slip by unnoticed.