Secure Coding Practices for Microsoft .NET Applications

White Paper

Amit Klein, Director of Security and Research

Sanctum, Inc.

1.0 Overview

As .NET applications explode with Web services adoption, security plays a critical role in the implementation of business operations based on these new technologies. This paper will detail the tenants of secure coding specific to .NET applications and give specific suggestions for the Visual Studio .NET environment. Specifically, this paper will focus on five common ASP.NET application security flaws, and recommendations for delivering higher quality applications.

2.0 The tenants of secure coding

2.1 Distrust relationship

The primal sin of all web applications is their tendency to trust user input. It is assumed that since browsers are used to interact with the site, then users – good and bad - are bound by the browser, and can only send data from the browser. This is obviously not true. It is amazingly easy to send any kind of data to the application. In fact, hackers have a rich toolkit of programs whose sole purpose is to provide a means to interact and attack sites outside the boundaries of the browser. From the lowest raw line-mode interface (e.g. telnet), through CGI scanners, web proxies, and web application scanners, attackers have a diverse spectrum of possible attack modes and means.

The only way to counteract the plethora of attack directions, techniques, and tools is to validate user input. Always, all input, all of the time, again and again. Here are some guidelines:

1. **Assume nothing on user input**
2. **Formulate your validation criteria for all user input**
3. **Enforce the validation criteria on all user input**
4. **Validate the data on a *trusted* machine (the server)**
5. **Trust only what you validated**
6. **Use multiple-tier validations**

Notes:

- Regarding guideline number 4, it goes without saying that the validation should take place on the server, a trusted platform, as opposed to on the client /browser which cannot be trusted. Client side JavaScript code that validates user input prior to submitting is a nice idea as far as performance and user experience, but from a security point of view, it's meaningless or even worse – it may provide a false sense of security. Anything that runs at the client side can be fooled, and it's especially easy to do so with Javascript code.

- Regarding guideline number 6, it makes sense to perform several, perhaps overlapping validations. For instance, a program may validate all input upon receiving it to make sure it consists of valid characters, and that no field is too long (potential buffer overflow). Some routines may then carry out further validations, making sure that the data is reasonable and valid for the specific purposes it will be used for. A more fine-grained character set validation may be applied, as well as length restriction enforcement.

2.2 Positive Thinking

The second tenant of secure coding is to formulate the validation in a positive manner. That is, to provide positive security, rather than negative security. Positive security means that only data known to be good is allowed into the system. Unknown, unrecognized or evil data is rejected. Negative security means that only data known to be evil is rejected, while other data, including unrecognized or unknown is allowed.

For example, an input field consisting of user name can be checked for characters that are allowed to be in a user name (e.g. alphanumeric characters) – this provides positive security. On the other hand, the input field can be checked for hazardous characters such as an apostrophe, or for forbidden patterns such as double hyphen– this provides negative security.

2.3 Comparison between positive security and negative security

	Positive Security	**Negative Security**
Definition	All data allowed into the system	All data disallowed into the system
Typical implementation	Allowed value list, allowed characters	Forbidden patterns, forbidden characters
Example - allowing valid file name (or blocking malicious file names via a pattern)	[a-zA-Z0-9]{1,20}\.html	\.\.\\|\\\.\.|\.\./|/\.\. (block the patterns "..\", "\..", "../", "/..")
Security	High – only valid data is allowed	Low – are all the hazardous characters listed? How can one be sure that the patterns suffice, and cannot be smartly bypassed?
Functionality	Relatively prone to blocking valid data	Less prone to blocking allowed data (although this can still happen, when patterns or forbidden characters are too broadly defined)

Obviously, when achieved, positive security, is superior to negative security, and should be used whenever possible.

3.0 "What's wrong with this picture?": Five common ASP.NET security flaws and suggested coding recommendations

3.1 Parameter tampering and ASP.NET field validators

a. The problem – parameter tampering

Trusting user input is the number one enemy of web application security. But how does this flaw appear in real life?

The major source for user input in a web application is the parameters submitted in HTML forms. Failing to validate these parameters may result in a severe security hole.

a. *Flawed code (C# querying a backend Microsoft SQL server, assuming the variables "user" and "password" are taken as-is from the user input)*

```
SqlDataAdapter my_query = new SqlDataAdapter(
      "SELECT * FROM accounts WHERE acc_user='" + user +
      "' AND acc_password='" + password + '"', the_connection);
```

Note:

- The code and examples throughout this paper work for MS-SQL servers, though the ideas are relevant practically to all database servers.

b. The result

While this looks relatively innocent, it in fact opens the gate to a most vicious SQL injection attack. By choosing the input field "user" to be `' OR 1=1--` the attacker can probably log-in into the system as an arbitrary user. A refinement of this is (assuming the attacker knows that the super-user's user name is "admin") to inject the data `admin' --` as the user field, in which case the attacker will be logged in as the super-user. And finally, it may be possible to execute shell commands, simply by appending the appropriate call right after the query, as in `'; EXEC master..xp_cmdshell('`*`shell command here`*`')--`

What's going on here? The programmer assumed that the user input consists of solely "normal" data – real user names, real passwords. These usually do not contain the character ' (apostrophe), which happens to play a major role on SQL's syntax. Therefore, there's no harm in generating the SQL query using valid data. But if the data is invalid and contains unexpected characters, such as ', then the query generated is not the query the programmer intended to execute, and therein lies the attack.

d. The solution: ASP.NET validators

Perhaps the most important contribution to ASP.NET's web application security is the introduction of field validators. A field validator, as the name hints, is a mechanism that enables the ASP.NET programmer to enforce some restrictions on the field value, thereby validating the field.

There are several types of field validators. In this case, we can use a regular expression validator (i.e. we use a validator that enforces that the user input field matches a given regular expression). In order to block the attack shown above, we need to forbid the apostrophe character, thus taking the negative security approach – `"[^']*"`. Better yet, we can formulate a regular expression that allows only alphanumeric characters for this field (thus taking the positive security approach) – `"[a-zA-Z0-9]*"`.

By incorporating and correctly using the field validator mechanism, the developer can programmatically secure all input fields of the application against attacks such as cross site scripting and SQL injection.

Further Reading:

- "User Input Validation in ASP.NET" - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/pdc_userinput.asp

- "Web Forms Validation" - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vboriWebFormsValidation.asp

- "ASP.NET validation in depth" - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/aspplusvalid.asp

3.2 Parameter tampering revisited - avoid validator pitfalls (and a note about information exposure)

a. The problem – parameter tampering (take II)

After reading the above section about ASP.NET field validators, you incorporate validators for every user input field. While you feel you should be safe from parameter tampering, sadly, you are not. How come? There are several pitfalls to the implementation of field validators; here are the important ones:

The first example demonstrates the importance of understanding the processing flow of ASP.NET pages with respect to field validators and error handling.

b. Flawed Code #1

```
<%@ Page Language="vb" %>

<form method="post" runat="server" ID="Form1">
Please Login<br>

User Name:
   <asp:Textbox ID="user" runat="server"/><br>
      <asp:RegularExpressionValidator
            ControlToValidate="user"
            ValidationExpression=
            "[a-zA-Z0-9]{1,10}"
            runat="server" />
Password:
   <asp:Textbox ID="pass" runat="server"/><br>
      <asp:RegularExpressionValidator
            ControlToValidate="pass"
            ValidationExpression=
            "[a-zA-Z0-9]{1,10}"
            runat="server" />

   <asp:Button id="cmdSubmit" runat="server" Text="Submit!"
OnClick="do_login"></asp:Button>
</form>

<script runat="server">
Sub do_login(sender As Object, e As System.EventArgs)
   ' I'm validated, so let's query the database
   …
End Sub
</script>
```

c. Result

The hacker can ignore the whole security mechanism - the character set validation code, since it does not actually affect the flow of the code. The hacker can, therefore, run SQL injection attacks just as described above.

d. Solution: Field validators must be explicitly checked

It is not enough to just define a validator for the field in question. Doing so indeed results in an error message in the HTML sent to the client, *as well as* the whole page being rendered, and processing not stopping once the validator failed. The right approach is to explicitly verify that the validator returned a positive result (logical “true”) before proceeding with processing the page and executing sensitive transactions. Verification of the validation can be done per validator, by querying the IsValid property of the validator. Alternatively, the logical AND of all validators is represented by the page property IsValid, which may be queried to get the success of all validators together.

A secure version of the above code would be:

```
<%@ Page Language="vb" %>

<form method="post" runat="server" ID="Form1">
Please Login<br>

User Name:
   <asp:Textbox ID="user" runat="server"/><br>
      <asp:RegularExpressionValidator
            ControlToValidate="user"
            ValidationExpression=
            "[a-zA-Z0-9]{1,10}"
            runat="server" />
Password:
   <asp:Textbox ID="pass" runat="server"/><br>
      <asp:RegularExpressionValidator
            ControlToValidate="pass"
            ValidationExpression=
            "[a-zA-Z0-9]{1,10}"
            runat="server" />

   <asp:Button id="cmdSubmit" runat="server" Text="Submit!"
OnClick="do_login"></asp:Button>
</form>

<script runat="server">
Sub do_login(sender As Object, e As System.EventArgs)
   If Page.IsValid Then
         ' I'm validated, so let's query the database
         …
   Else
         … error handing
   End If
End Sub
</script>
```

Further reading:

- "Testing Validity Programmatically for Asp.NET Server Controls" http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbtsktestingvalidityprogrammatically.asp

The second example is about the correct syntax and usage of the RangeValidator.

e. Flawed Code #2

```
<!-- check for a number 1-9 -->
<asp:RangeValidator … MinimumValue="1" MaximumValue="9" …/>
```

f. The result

The hacker can actually enter any positive number to the application (e.g. “123”), as well as some non-numeric data (e.g. “0abcd”). The application may enter an undefined state.

g. The solution: Range validation should specify the correct data type

When using the range validator ASP.NET control, it is important to keep in mind that the Type attribute must be set according to the type of input field expected. The Type attribute defaults to “String”. This has a nasty consequence if the developer forgets about it or is unaware to it, as we saw in the above flawed code. Since no Type is specified, ASP.NET assumed “String”, meaning that the order is a lexicographical one. Therefore, the validator will only ensure that the string starts with 0-9. Strings such as “0abcd” will be accepted.

The right way to test for integer range is to specify the type as “Integer”, e.g.:

```
<!-- check for a number 1-9 -->
<asp:RangeValidator … MinimumValue="1" MaximumValue="9"
Type="Integer" … />
```

Further reading:

- “RangeValidator Control” - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconrangevalidatorcontrol.asp

The third example is about an easy to miss shortcoming of performing client side verification:

h. Flawed code #3

```
<asp:RegularExpressionValidator
      ControlToValidate="user"
      ValidationExpression=
            "Jim|Joe|Charlie|Admin|System|Frank" …
/>
```

i. The result

The attacker gains valuable information – the names of the admin accounts. While this may not be useful for this page (after all, this particular value is allowed), it may be of use in other pages.

This happens since by default, the validator code is executed both at the client side and at the server side. The client side code provides good performance since there is no need for the request to be sent to the server and the response to be returned, and a good user experience with immediate validation before the data is actually sent. The server side code provides security (validation on a trusted machine). The downside to this scheme is that the security validation parameters are exposed to the client, since the same validation is run there. In some cases, this has a negative overall effect.

For example, a system that is designed to let in only certain users through its login page may have a regular expression validator for the user name such as "Jim|Joe|Charlie|Admin|System|Frank". This is definitely the best one can get along the lines of positive security (only the designated 6 usernames are valid), however, since by default the validation is also performed at the client side, this information will be found in the HTML page presented to the client. And consequently, the client may be able to reverse engineer the validator, and learn the name of the (only) 6 valid accounts.

j. The solution

Either disable validating at the client side for validators that may expose sensitive information (this can be done by setting the EnableClientScript property of the validator control to "false"), and/or validate this data using a different mechanism.

The below secure code takes the first approach – validation is carried out at the server side only:

```
<asp:RegularExpressionValidator
      ControlToValidate="user"
      ValidationExpression=
            "Jim|Joe|Charlie|Admin|System|Frank"
      EnableClientScript="False" …
/>
```

Further reading

- "Disabling Client Side Validation" - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbtskdisablingvalidationatruntime.asp

3.3 Information leakage: Remember that __VIEWSTATE data can be viewed

a. The problem: information about the application internals leaks out

An often overlooked source for information about ASP.NET application is the __VIEWSTATE hidden field, which can be found in almost all HTML pages. This hidden field is overlooked because it is Base64 encoded, which makes it look like an innocent string of alphanumeric characters (actually, forward slash, plus sign and equal sign are also part of the Base64 character set).

b. Flawed code (the web.config configuration file)

```
<configuration>
      …
   <system.web>
         …  (no <machineKey> element)
   </system.web>
…
</configuration>
```

c. Result

The __VIEWSTATE's Base64 encoding can be easily decoded, and the __VIEWSTATE data can be exposed with minimal effort. Now the attacker can see the information that may be sensitive, such as internal state data of the application.

By default, the __VIEWSTATE data consists of:

- Dynamic data from page controls
- Data stored explicitly by the developer in the ViewState bag
- Cryptographic signature of the above data

The first two data items appear in the clear, and as such provide an attacker with information about the application. The third item, the cryptographic signature, ensures that the data cannot be tampered with, yet the data itself is not encrypted.

d. The solution: encrypt the __VIEWSTATE data

```
<configuration>
      …
   <system.web>
         …
            <machineKey validation="3DES"/>
            …
   </system.web>
…
</configuration>
```

Further reading:

- "Taking a Bite Out of ASP.NET ViewState" - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspnet/html/asp11222001.asp

3.4 SQL injection - Use SQL parameters to prevent SQL injection

a. The problem: SQL injection

The problem was described in the section "parameter tampering" above. Reminder: an SQL query was formed by the script by embedding user input. A malicious character (apostrophe), when placed in the input field, caused the SQL server to execute a query totally different than the one intended.

b. Flawed code

```
SqlDataAdapter my_query = new SqlDataAdapter(
      "SELECT * FROM accounts WHERE acc_user='" + user +
      "' AND acc_password='" + password + '"',
      the_connection);
```

c. The result

Just like the first example, by inserting the apostrophe character, an attacker can completely change the meaning of the SQL query. Consequently, an attacker can shape his/her own query, run different additional queries, and possibly execute SQL commands, which may compromise the server.

d. The solution

The obvious solution is to allow only the characters that are really needed. But what if apostrophe is in fact needed? In some cases, an apostrophe can be part of a person's name, or part of a perfectly valid English sentence.
The more robust approach to SQL injection prevention is the use of SQL parameters API (such as provided by ADO.NET) in order to have the programming infrastructure, and not the programmer, construct the query.

Using such an API, the programmer needs to provide a template query or a stored procedure, and a list of parameter values. These parameters are then securely embedded into the query and the result is executed by the SQL server. The advantage is in the process of embedding the parameters by the infrastructure, since it is guaranteed that the parameters will be embedded correctly. For example, apostrophes will be escaped, thus rendering SQL injection attacks useless.

So instead of the code in the “parameter tampering” section, use:

```
SqlDataAdapter my_query = new SqlDataAdapter(
      "SELECT * FROM accounts WHERE acc_user= @user AND
      acc_password=@pass", the_connection);

SqlParameter userParam =
      my_query.SelectCommand.Parameters.Add(
      "@user",SqlDbType.VarChar,20);
userParam.Value=user;

SqlParameter passwordParam =
      my_query.SelectCommand.Parameters.Add(
      "@pass",SqlDbType.VarChar,20);
passwordParam.Value=password;
```

This ensures that the apostrophe character is properly escaped, and will not jeopardize the application or the SQL database. At the same time, the apostrophe will not be blocked, which is an upside of this approach.

Further reading:

- “Data Access Security” (see the section “SQL Injection Attacks”) http://msdn.microsoft.com/library/en-us/dnnetsec/html/SecNetch12.asp?frame=true#sqlinjectionattacks

- “Building SQL Statements Securely” - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csvr2002/htm/cs_se_securecode_pajt.asp

3.5 Cross Site Scripting (insecure composition of HTML pages) – HTML encode outgoing data

a. The problem: Cross Site Scripting

An application vulnerable to cross-site scripting is one that embeds malicious user input to the response (HTML) page. To learn more about cross-site scripting attacks, it is suggested that you read the paper “Cross Site Scripting Explained”at www.sanctuminc.com/pdf/WhitePaper_CSS_Explained.pdf *b. Flawed code*

```
<%@ Page Language="vb" %>

<asp:Label id="Label1" runat="server">INITIAL LABEL
VALUE</asp:Label>

<form method="post" runat="server" ID="Form1">
Please Provide feedback<br>
   <asp:Textbox ID="feedback" runat="server"/><br>
   <asp:Button id="cmdSubmit" runat="server" Text="Submit!"
OnClick="do_feedback"></asp:Button>
</form>

<script runat="server">
Sub do_feedback(sender As Object, e As System.EventArgs)
   Label1.Text=feedback.Text
End Sub
</script>
```

c. The result

An attacker can form a malicious request with JavaScript code that will get executed at the client browser whe n the link is clicked. To see that this is possible, the above script can be fed with the following input:

```
<script>alert(document.cookie)</script>
```

d. The solution: HTML-encode user data that is sent back in the HTML response

On top of user input validation (in this case, does a normal user have to use the less-than symbol and the greater-than symbol? Perhaps these can be considered invalid characters), the classic solution to this problem is to HTML-encode outgoing user data. HTML-encoding of the data presented in the HTML page ensures that this data is not interpreted (by the browser) as anything other than plain text. Thus, the script injection attack is completely de-fanged.

In the above case, this maps simply to adding a function call to HtmlEncode in one place:

```
   …
   Label1.Text=Server.HtmlEncode(feedback.Text)
   …
```

As a result, the response HTML stream will contain:

```
   &lt;script&gt;alert(document.cookie)&lt;/script&gt;
```

Which is indeed harmless – no Javascript code is executed by the browser, because no HTML "script" tag is present. The less-than symbol and greater-than symbol are replaced by their HTML-encoded version, `&lt;` and `&gt;` respectively.

Note that ideally, this method should be combined with user input validation, thus providing a two tiers security architecture for the application.

Further reading:

- "Cross Site Scripting Explained" - http://www.sanctuminc.com/pdf/WhitePaper_CSS_Explained.pdf

- "Security Tips: Defend Your Code with Top Ten Security Tips Every Developer Must Know" (see tip #3 – "Prevent Cross-Site Scripting") - http://msdn.microsoft.com/msdnmag/issues/02/09/SecurityTips/default.aspx

- "HttpServerUtility.HtmlEncode method" (documentation of the HtmlEncode function) - http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemWebHttpServerUtilityClassHtmlEncodeTopic.asp?frame=true

Note:
The documentation for HtmlEncode is identical to that of UrlEncode – this seems to be a mistake in HtmlEncode's documentation.

4.0 Conclusion

ASP.NET provides several exciting productivity and security features, but these should be understood and used wisely. Failing to use the ASP.NET functions properly results in an insecure web application. We see therefore that ASP.NET does not exempt the programmer from following coding standards and procedures in order to write safe and secure applications.

The ASP.NET coding standards recommended in this paper are:

1. **Using ASP.NET validators to validate user input**
2. **Defining and using validators correctly (avoiding pitfalls and shortcomings of validators)**
3. **Encrypting the __VIEWSTATE**
4. **Using SQL parameters to form SQL queries from user data**
5. **Embedding user data as HTML only after HTML-encoding it**

In order to verify the programmer's adherence to secure coding practices, automatic testing of the application's vulnerability to web application attacks is needed. With the use of an automated security testing tool, this should take place as part of the development process to reduce the costs associated with fixing issues that are raised as a result of the testing. And by associating the security problem with the appropriate remedy, and having the programmer react immediately to the problem, the programmer is also undergoing an educational process, which can reduce the likelihood of him/her coding the same mistake again.

To conclude, understanding the recommended coding standards, augmented by using an automatic Web Application Security tool to test the adherence of the code to the standards, results in a systematic bug catching process, shorter find-fix cycles, and an easier learning curve for programmers. This in turn ensures shorter time to market, which is a key for the success of any development organization.

Further reading:

- Developing Secure Web Applications Just Got Easier: http://www.sanctuminc.com/pdf/WhitePaper_DevSecureAppsJustGotEasier.pdf

5.0 Acknowledgement

The section titled “parameter tampering revisited” is partially based on research conducted together with Ory Segal and Chaim Linhart (both from Sanctum Inc.).

For more information:
Sanctum Inc.
www.SanctumInc.com
sanctumsales@sanctuminc.com
phone: 877-888-3970 (US/Canada)
408-352-2000 (International)
+44 7710949512 (European HQ)