

Top 10 Security Vulnerabilities in .NET Configuration Files

Are your web applications vulnerable?

By Bryan Sullivan

Top 10 Security Vulnerabilities in .NET Configuration Files

Table of Contents

<i>Introduction</i>	3
<i>1. Custom Errors Disabled</i>	4
<i>2. Leaving Tracing Enabled</i>	8
<i>3. Debugging Enabled</i>	12
<i>4. Cookies Accessible Through Client-Side Script</i>	14
<i>5. Cookieless Session State Enabled</i>	16
<i>Intermission</i>	18
<i>6. Cookieless Authentication Enabled</i>	19
<i>7. Failure to Require SSL for Authentication Cookies</i>	20
<i>8. Sliding Expiration Used</i>	22
<i>9. Non-Unique Authentication Cookie Used</i>	23
<i>10. Hardcoded Credentials Used</i>	24
<i>About SPI Labs</i>	29
<i>Contact Information</i>	30

Top 10 Security Vulnerabilities in .NET Configuration Files

Introduction

These days, the biggest threat to an organization's network security comes from its public web site. Unlike internal-only network services such as databases—which can be sealed off from the outside via firewalls—a public web site is generally accessible to anyone who wants to view it. As networks have become more secure, vulnerabilities in web applications have inevitably attracted the attention of hackers, both criminal and recreational, who have devised techniques to exploit these holes. In fact, attacks upon the web application layer now exceed those conducted at the network level, and can have consequences which are just as damaging.

Some enlightened software architects and developers are becoming educated on these threats and are designing their web applications with security in mind. By “baking in” security from the start of the development process rather than trying to “brush it on” at the end, you are much more likely to create secure applications that will withstand hackers' attacks. However, even the most meticulous and security-aware C# or VB.NET code can still be vulnerable to attack if you neglect to secure the configuration files of your application. An incorrectly configured application can be just as dangerous as an incorrectly coded one. To make matters worse, many configuration settings *default* to insecure values.

An additional problem is that `web.config` files were designed to be changed at any time, even after the application is in production. A well-intentioned

Top 10 Security Vulnerabilities in .NET Configuration Files

system administrator could inadvertently open the web site to attack just by modifying the configuration file. And because .NET configuration files operate in a hierarchical manner, a single change to the global `Machine.config` file could affect every web site on the entire network. This white paper lists ten of the “worst offenders” of security misconfigurations, and demonstrates their potential impact on your applications. It also provides some best practices for locking down your configuration files to ensure that they are not unintentionally modified by well-meaning (but uninformed) programmers or administrators.

1. Custom Errors Disabled

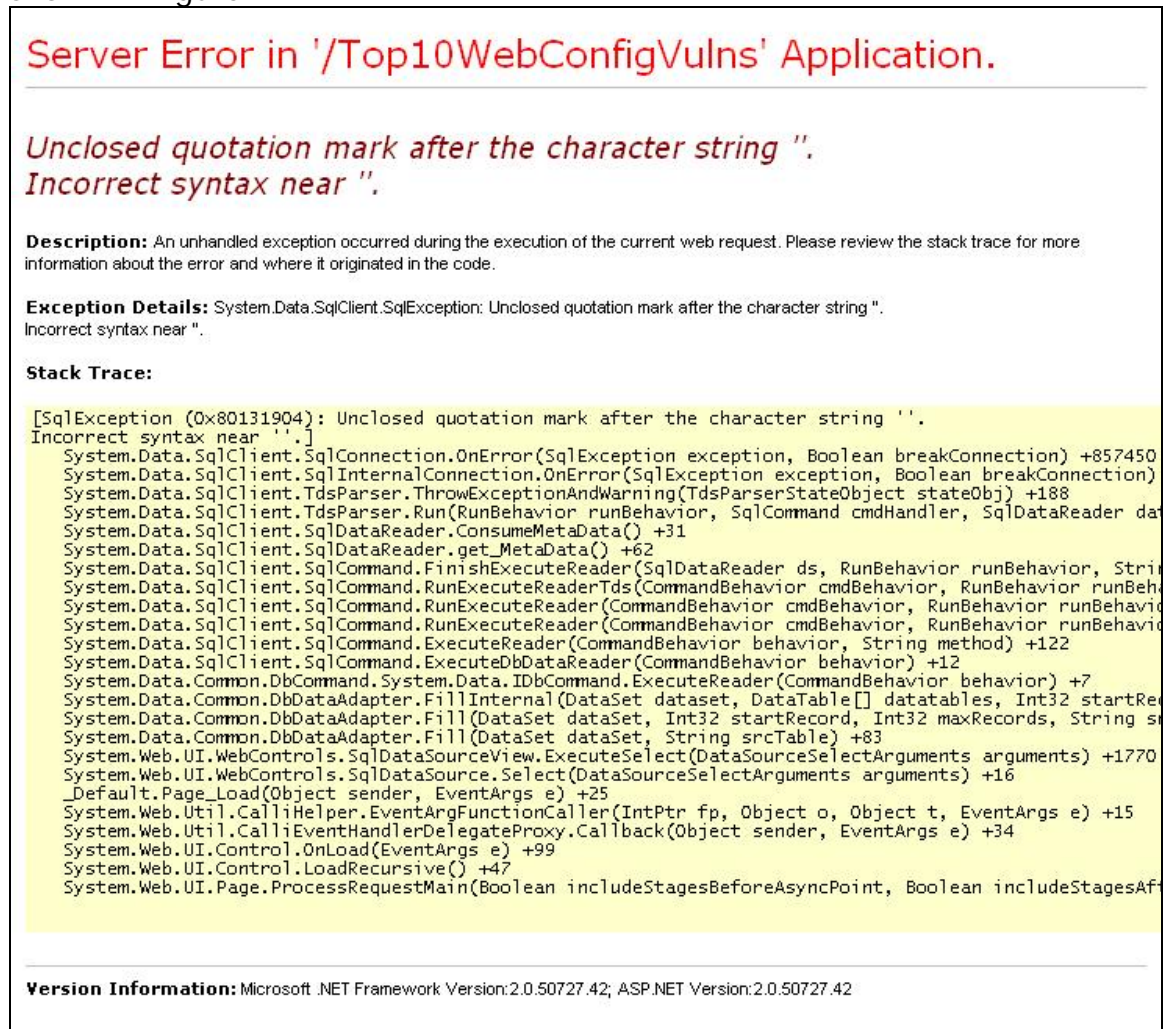
When you disable custom errors as shown below, ASP.NET provides a detailed error message to clients by default.

Vulnerable:	Secure:
<pre><configuration> <system.web> <customErrors mode="Off" ></pre>	<pre><configuration> <system.web> <customErrors mode="RemoteOnly" ></pre>

In itself, knowing the source of an error may not seem like a security risk. However, consider this: the more information a hacker can gather about a web site, the more likely it is that he will be able to successfully attack it. For example, the error message lists the web server being used, the operating system, database types, etc. All of these bits of data help hackers trying to

Top 10 Security Vulnerabilities in .NET Configuration Files

compromise your application. An error message can be a gold mine of information to an attacker. For example, take the sample ASP.NET error page shown in Figure 1.



Server Error in '/'Top10WebConfigVulns' Application.

Unclosed quotation mark after the character string ''.
Incorrect syntax near ''.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Unclosed quotation mark after the character string ''.
Incorrect syntax near ''.

Stack Trace:

```
[SqlException (0x80131904): Unclosed quotation mark after the character string ''.  
Incorrect syntax near ''.]  
System.Data.SqlClient.SqlConnection.OnError(SqlException exception, Boolean breakConnection) +857450  
System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, Boolean breakConnection)  
System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject stateObj) +188  
System.Data.SqlClient.TdsParser.Run(RunBehavior runBehavior, SqlCommand cmdHandler, SqlDataReader data  
System.Data.SqlClient.SqlDataReader.ConsumeMetaData() +31  
System.Data.SqlClient.SqlDataReader.get_MetaData() +62  
System.Data.SqlClient.SqlCommand.FinishExecuteReader(SqlDataReader ds, RunBehavior runBehavior, Stri  
System.Data.SqlClient.SqlCommand.RunExecuteReaderTds(CommandBehavior cmdBehavior, RunBehavior runBeh  
System.Data.SqlClient.SqlCommand.RunExecuteReader(CommandBehavior cmdBehavior, RunBehavior runBehav  
System.Data.SqlClient.SqlCommand.RunExecuteReader(CommandBehavior cmdBehavior, RunBehavior runBehav  
System.Data.SqlClient.SqlCommand.ExecuteReader(CommandBehavior behavior, String method) +122  
System.Data.SqlClient.SqlCommand.ExecuteReader(CommandBehavior behavior) +12  
System.Data.Common.DbCommand.System.Data.IDbCommand.ExecuteReader(CommandBehavior behavior) +7  
System.Data.Common.DbDataAdapter.FillInternal(DataSet dataset, DataTable[] datatables, Int32 startRe  
System.Data.Common.DbDataAdapter.Fill(DataSet dataSet, Int32 startRecord, Int32 maxRecords, String s  
System.Data.Common.DbDataAdapter.Fill(DataSet dataSet, String srcTable) +83  
System.Web.UI.WebControls.SqlDataSourceView.ExecuteSelect(DataSourceSelectArguments arguments) +1770  
System.Web.UI.WebControls.SqlDataSource.Select(DataSourceSelectArguments arguments) +16  
_Default.Page_Load(Object sender, EventArgs e) +25  
System.Web.Util.CalliHelper.EventArgFunctionCaller(IntPtr fp, Object o, Object t, EventArgs e) +15  
System.Web.Util.CalliEventHandlerDelegateProxy.Callback(Object sender, EventArgs e) +34  
System.Web.UI.Control.OnLoad(EventArgs e) +99  
System.Web.UI.Control.LoadRecursive() +47  
System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean includeStagesAff
```

Version Information: Microsoft .NET Framework Version:2.0.50727.42; ASP.NET Version:2.0.50727.42

Figure 1. Detailed Error Message. From the information shown in this error message, an attacker can discover the ASP.NET and .NET framework versions, that the application uses SQL Server, and that the application may be vulnerable to SQL injection attacks.

Top 10 Security Vulnerabilities in .NET Configuration Files

By simply looking at the error page in Figure 1, an attacker can see that the application is using the .NET framework version 2.0.50727.42 and ASP.NET version 2.0.50727.42. Knowing that, he can search the web for known security advisories concerning these specific product versions. Also, knowing that the application uses ASP.NET 2.0 tells him that the server is running a recent version of Microsoft Windows (either XP or Server 2003) and that Microsoft Internet Information Server (IIS) 6.0 or later is being used as the web server. This leads to more searching for security advisories. The hacker can also determine that the application is using Microsoft SQL Server as its database because the exception type thrown was a `SqlException`, which is specific to Microsoft SQL Server. Finally, the real prize for the hacker is the exception message itself, which tells him that the command being sent to the database was created in an ad-hoc fashion—and therefore that the page is likely vulnerable to SQL injection attacks.

You can prevent such information leakage by modifying the `mode` attribute of the `<customErrors>` element to `On` or `RemoteOnly`. This setting instructs the web application to display a nondescript, generic error message when an unhandled exception is generated (see Figure 2).

Top 10 Security Vulnerabilities in .NET Configuration Files

Server Error in '/Top10WebConfigVulns' Application.

Runtime Error

Description: An application error occurred on the server. The current custom error settings for this application prevent the details of the application error from being viewed.

Details: To enable the details of this specific error message to be viewable on the local server machine, please create a <customErrors> tag within a "web.config" configuration file located in the root directory of the current web application. This <customErrors> tag should then have its "mode" attribute set to "RemoteOnly". To enable the details to be viewable on remote machines, please set "mode" to "Off".

```
<!-- Web.Config Configuration File -->
<configuration>
  <system.web>
    <customErrors mode="RemoteOnly"/>
  </system.web>
</configuration>
```

Notes: The current error page you are seeing can be replaced by a custom error page by modifying the "defaultRedirect" attribute of the application's <customErrors> configuration tag to point to a custom error page URL.

```
<!-- Web.Config Configuration File -->
<configuration>
  <system.web>
    <customErrors mode="On" defaultRedirect="mycustompage.htm"/>
  </system.web>
</configuration>
```

Figure 2. Generic Error Message: Switching the <customErrors> element's mode attribute to On or RemoteOnly causes the server to generate a generic error message and discloses far less potentially sensitive security information.

Another way to circumvent the problem is to create your own custom error page and redirect users to that page when errors occur. You can specify the page by setting the defaultRedirect attribute of the <customErrors> element. This approach can provide even better security because the default generic error page shown in Figure 2 still gives away too much information about the system (namely, that it's using a web.config file, which reveals that the server is running ASP.NET). Your custom page can be as generic and

Top 10 Security Vulnerabilities in .NET Configuration Files

unspecific as you like, and should not reveal any details of the application or the server environment.

2. Leaving Tracing Enabled

The trace feature of ASP.NET is one of the most useful tools that you can use to debug and profile your applications. Unfortunately, it is also one of the most useful tools that a hacker can use to attack your applications if tracing is left enabled in a production environment.

Vulnerable:	Secure:
<pre><configuration> <system.web> <trace enabled="true" localOnly="false" ></pre>	<pre><configuration> <system.web> <trace enabled="false" localOnly="true" ></pre>

When the `<trace>` element is enabled for remote users (`localOnly="false"`), any user can view an incredibly detailed list of recent requests to the application simply by browsing to the page `trace.axd`. They'll see a page similar to Figure 3 and Figure 4.

Top 10 Security Vulnerabilities in .NET Configuration Files

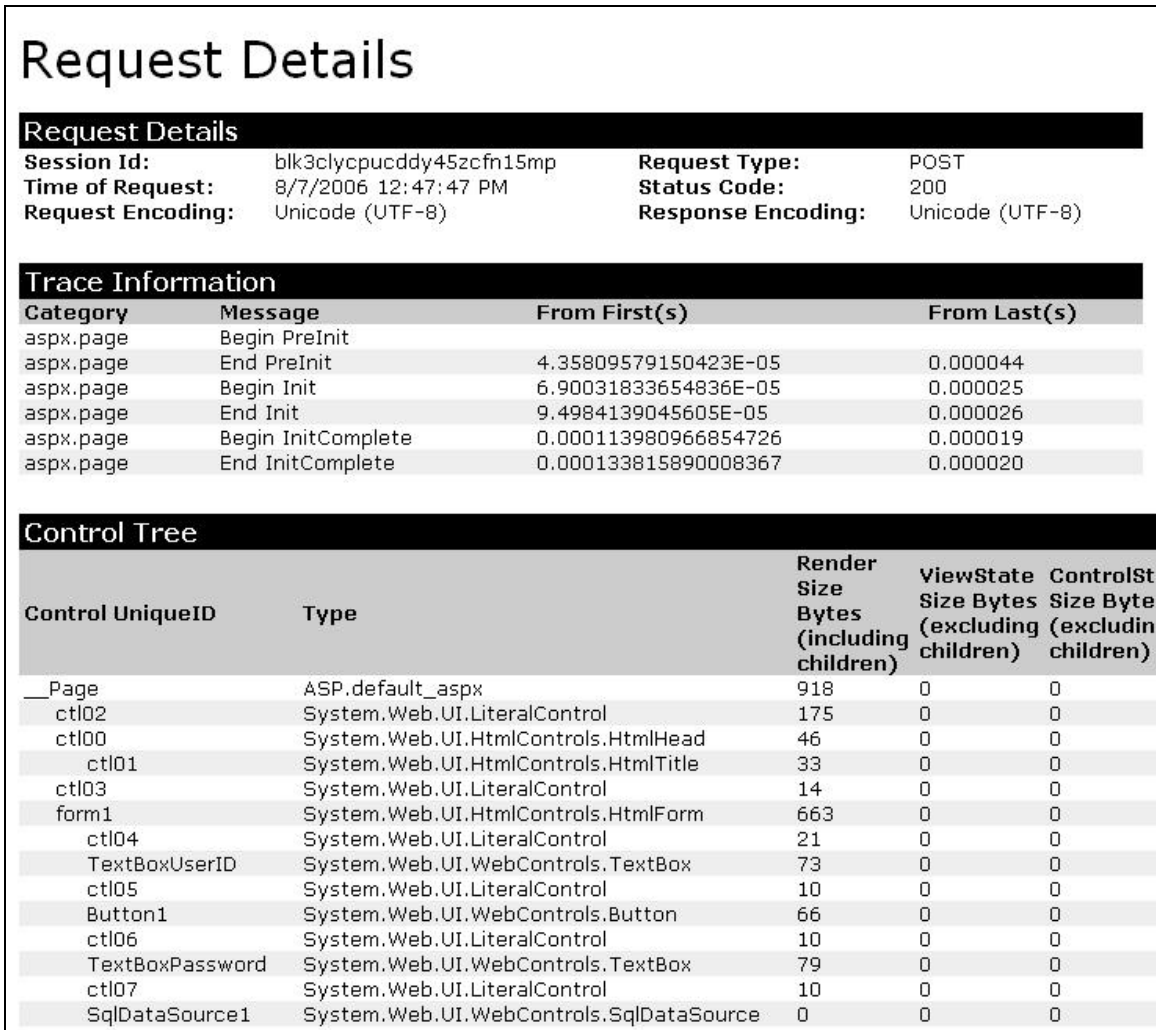


Figure 3. Trace.axd Output: This detailed Trace output contains complete request information, including the values of form and server variables.

Top 10 Security Vulnerabilities in .NET Configuration Files

Accept	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
Accept-Encoding	gzip, deflate
Accept-Language	en-us
Host	localhost
Referer	http://localhost/Top10WebConfigVulns/Default.aspx
User-Agent	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.0.3705; .NET CLR 1.1.4343.2)
Form Collection	
Name	Value
__VIEWSTATE	/wEPDwULLTE5MzExNTg0MThkZAEOQjY36SZYZt9tQmHVEm569kPQ
TextBoxUserID	bob
Button1	Button
TextBoxPassword	Elvis
__EVENTVALIDATION	/wEWBALT1sXcBgLD57fLAQKM54rGBgKpzpH0DZnsyOcsFyxozuH6pNjcpsyIEnkF
QueryString Collection	
Name	Value
Server Variables	
Name	Value
ALL_HTTP	HTTP_CACHE_CONTROL:no-cache HTTP_CONNECTION:Keep-Alive HTTP_CONTENT_ENCODING:application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */* HTTP_ACCEPT_LANGUAGE:en-us HTTP_HOST:localhost HTTP_REFERER:http://localhost/Top10WebConfigVulns/Default.aspx HTTP_USER_AGENT:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.0.3705; .NET CLR 1.1.4343.2) Cache-Control: no-cache Connection: Keep-Alive Content-Length: 206 Content-Type: application/msword
ALL_RAW	Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */* Accept-Encoding: gzip, deflate Accept-Language: en-us Referer: http://localhost/Top10WebConfigVulns/Default.aspx User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.0.3705; .NET CLR 1.1.4343.2)
APPL_MD_PATH	/LM/w3svc/1/ROOT/Top10WebConfigVulns
APPL_PHYSICAL_PATH	c:\inetpub\wwwroot\Top10WebConfigVulns\
AUTH_TYPE	
AUTH_USER	

Figure 4. Trace.axd Output (continued): This detailed Trace output contains complete request information, including the values of form and server variables.

If a detailed exception message such as Figure 1 is like a gold mine to a hacker, trace logs as shown in Figure 3 and Figure 4 are like Fort Knox! Just

Top 10 Security Vulnerabilities in .NET Configuration Files

look at the wealth of information: the .NET and ASP.NET versions that the server is running; a complete trace of all the page methods that the request caused, including their times of execution; the session state and application state keys; the request and response cookies; the complete set of request headers, form variables, and QueryString variables; and finally the complete set of server variables.

A hacker would obviously find the form variables useful because these might include email addresses that could be harvested and sold to spammers, IDs and passwords that could be used to impersonate the user, or credit card and bank account numbers. Even the most innocent-looking piece of data in the trace collection can be dangerous in the wrong hands. For example, the `APPL_PHYSICAL_PATH` server variable, which contains the physical path of the application on the server, could help an attacker perform directory traversal attacks against the system.

The best way to prevent a hacker from obtaining trace data is to disable the trace viewer completely by setting the `enabled` attribute of the `<trace>` element to `false`. If you must have the trace viewer enabled, either to debug or to profile your application, then be sure to set the `localOnly` attribute of the `<trace>` element to `true`. This setting allows users to access the trace viewer only from the web server and disables viewing it from any remote machine.

Top 10 Security Vulnerabilities in .NET Configuration Files

3. Debugging Enabled

Deploying a web application in debug mode is a very common mistake. Virtually every web application requires some debugging. Visual Studio 2005 will even automatically modify the web.config file to allow debugging when you start to debug your application. And, since deploying ASP.NET applications is as simple as copying the files from the development folder into the deployment folder, it's easy to see how development configuration settings can accidentally make it into production.

Vulnerable:	Secure:
<pre><configuration> <system.web> <compilation debug="true"></pre>	<pre><configuration> <system.web> <compilation debug="false"></pre>

Like the first two security vulnerabilities described in this list, leaving debugging enabled is dangerous because you are providing inside information to end users who shouldn't have access to it, and who may use it to attack your applications. For example, if you have enabled debugging and disabled custom errors in your application, then any error message displayed to an end user will include not only the server information, a detailed exception message, and a stack trace, but also the actual source code of the page where the error occurred (see Figure 5).

Top 10 Security Vulnerabilities in .NET Configuration Files

Server Error in '/Top10WebConfigVulns' Application.

*Unclosed quotation mark after the character string '''.
Incorrect syntax near '''.*

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Unclosed quotation mark after the character string '''.
Incorrect syntax near '''.

Source Error:

```
Line 14:     {  
Line 15:         this.SqlDataSource1.SelectCommand = "SELECT * FROM Customer WHERE CustomerID = '' + t  
Line 16:         this.SqlDataSource1.Select(DataSourceSelectArguments.Empty);  
Line 17:     }  
Line 18: }
```

Source File: c:\inetpub\wwwroot\Top10WebConfigVulns\Default.aspx.cs **Line:** 16

Stack Trace:

```
[SqlException (0x80131904): Unclosed quotation mark after the character string '''.  
Incorrect syntax near '''.]  
System.Data.SqlClient.SqlConnection.OnError(SqlException exception, Boolean breakConnection) +857450  
System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, Boolean breakConnection)  
System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject stateObj) +188  
System.Data.SqlClient.TdsParser.Run(RunBehavior runBehavior, SqlCommand cmdHandler, SqlDataReader dat  
System.Data.SqlClient.SqlDataReader.ConsumeMetaData() +31  
System.Data.SqlClient.SqlDataReader.get_MetaData() +62  
System.Data.SqlClient.SqlCommand.FinishExecuteReader(SqlDataReader ds, RunBehavior runBehavior, Stri  
System.Data.SqlClient.SqlCommand.RunExecuteReaderTds(CommandBehavior cmdBehavior, RunBehavior runBeh  
System.Data.SqlClient.SqlCommand.RunExecuteReader(CommandBehavior cmdBehavior, RunBehavior runBehav  
System.Data.SqlClient.SqlCommand.RunExecuteReader(CommandBehavior cmdBehavior, RunBehavior runBehav  
System.Data.SqlClient.SqlCommand.ExecuteReader(CommandBehavior behavior, String method) +122  
System.Data.SqlClient.SqlCommand.ExecuteReader(CommandBehavior behavior) +12  
System.Data.Common.DbCommand.System.Data.IDbCommand.ExecuteReader(CommandBehavior behavior) +7  
System.Data.Common.DbDataAdapter.FillInternal(DataSet dataset, DataTable[] datatables, Int32 startRe  
System.Data.Common.DbDataAdapter.Fill(DataSet dataSet, Int32 startRecord, Int32 maxRecords, String s  
System.Data.Common.DbDataAdapter.Fill(DataSet dataSet, String srcTable) +83
```

Figure 5. Detailed Error Message with Source Code: Leaving debugging enabled causes the server to output detailed information, including source code where the error occurred.

Unfortunately, this configuration setting isn't the only way that source code might be displayed to the user. Here's a story that illustrates how developers

Top 10 Security Vulnerabilities in .NET Configuration Files

can't concentrate solely on one type of configuration setting to improve security. In early versions of Microsoft's Atlas framework, some Atlas controls would return a stack trace with source code to the client browser whenever exceptions occurred. This behavior happened *regardless* of the custom error setting in the configuration. So, even if you properly configured your application to display non-descriptive messages when errors occurred, you could still have unexpectedly revealed your source code to your end users if you forgot to disable debugging.

To disable debugging, set the value of the `debug` attribute of the `<compilation>` element to `false`. This is the default value of the setting, but as we will see, it's safer to explicitly set the desired value rather than relying on the defaults.

4. Cookies Accessible Through Client-Side Script

In Internet Explorer 6.0, Microsoft introduced a new cookie property called `HttpOnly`. While you can set the property programmatically, you can set it generically in the site configuration.

Vulnerable:	Secure:
<pre><configuration> <system.web> <httpCookies httpOnlyCookies="false"></pre>	<pre><configuration> <system.web> <httpCookies httpOnlyCookies="true"></pre>

Top 10 Security Vulnerabilities in .NET Configuration Files

Any cookie marked with this property will be accessible only from server-side code, and not to any client-side scripting code like JavaScript or VBScript. This shielding of cookies from the client helps to protect the application from cross-site scripting attacks. A hacker initiates a cross-site scripting (XSS) attack by attempting to insert his script code into the web page. Any page that accepts input from a user and echoes that input back is potentially vulnerable. For example, a login page that prompts for a user name and password and then displays "Welcome back, <username>" on a successful login is possibly susceptible to an XSS attack.

Message board or forum pages are also often vulnerable. In these pages, legitimate users post their thoughts or opinions, which are then visible to all other visitors to the site. But an attacker, rather than posting about the current topic, will instead post a message such as "<script>alert(document.cookie);</script>". The message board now includes the attacker's script code in its page code—and the browser then interprets and executes it for future site visitors. Usually attackers use such script code to try to obtain the user's authentication token (usually stored in a cookie), which they could then use to impersonate the user. When cookies are marked with the `HttpOnly` property, their values are hidden from the client, so this attack will fail.

It is possible to enable `HttpOnly` programmatically on any individual cookie by setting the `HttpOnly` property of the `HttpCookie` object to `true`. However,

Top 10 Security Vulnerabilities in .NET Configuration Files

it is easier and more reliable to configure the application to automatically enable `HttpOnly` for all cookies. To do this, set the `httpOnlyCookies` attribute of the `<httpCookies>` element to `true`.

5. Cookieless Session State Enabled

In the initial 1.0 release of ASP.NET, you had no choice about how to transmit the session token between requests when your web application needed to maintain session state: it was always stored in a cookie. Unfortunately, this meant that users who would not accept cookies could not use your application. So, in ASP.NET 1.1, Microsoft added support for cookieless session tokens via use of the “cookieless” setting.

Vulnerable:	Secure:
<pre><configuration> <system.web> <sessionState cookieless="UseUri"></pre>	<pre><configuration> <system.web> <sessionState cookieless="UseCookies"></pre>

Web applications configured to use cookieless session state now stored the session token in the page URLs rather than a cookie. For example, the page URL might change from `http://myserver/MyApplication/default.aspx` to `http://myserver/MyApplication/(123456789ABCDEFGH)/default.aspx`. In this case, `123456789ABCDEFGH` represents the current user's session token. A different user browsing the site at the same time would receive a completely

Top 10 Security Vulnerabilities in .NET Configuration Files

different session token, resulting in a different URL, such as

```
http://myserver/MyApplication/(ZYXWVU987654321)/default.aspx.
```

While adding support for cookieless session state did improve the usability of ASP.NET web applications for users who would not accept cookies, it also had the side effect of making those applications much more vulnerable to session hijacking attacks. Session hijacking is basically a form of identity theft wherein a hacker impersonates a legitimate user by stealing his session token. When the session token is transmitted in a cookie, and the request is made on a secure channel (that is, it uses SSL), the token is secure. However, when the session token is included as part of the URL, it is much easier for a hacker to find and steal it. By using a network monitoring tool (also known as a “sniffer”) or by obtaining a recent request log, hijacking the user’s session becomes a simple matter of browsing to the URL containing the stolen unique session token. The web application has no way of knowing that this new request with session token 123456789ABCDEFG is not coming from the original, legitimate user. It happily loads the corresponding session state and returns the response back to the hacker, who has now effectively impersonated the user.

The most effective way to prevent these session hijacking attacks is to force your web application to use cookies to store the session token. This is accomplished by setting the `cookieless` attribute of the `<sessionState>` element to `UseCookies` or `false`. But what about the users who do not

Top 10 Security Vulnerabilities in .NET Configuration Files

accept cookies? Do you have to choose between making your application available to all users versus ensuring that it operates securely for all users? A compromise between the two is possible in ASP.NET 2.0. By setting the `cookieless` attribute to `AutoDetect`, the application will store the session token in a cookie for users who accept them and in the URL for those who won't. This means that only the users who use cookieless tokens will still be vulnerable to session hijacking. That's often acceptable, given the alternative—that users who deny cookies wouldn't be able to use the application at all. It is ironic that many users disable cookies because of privacy concerns when doing so can actually make them more prone to attack.

Intermission

The first five `web.config` vulnerabilities discussed in this white paper have been applicable to all ASP.NET web applications regardless of their methods of authentication, or even whether they use authentication at all. The next five apply only to applications using Forms authentication. If this applies to your web sites, please read on. If not, don't leave yet! After the top-ten list is complete, I'll show you some methods of locking down your configuration files so that they can't be modified unintentionally.

Top 10 Security Vulnerabilities in .NET Configuration Files

6. Cookieless Authentication Enabled

Just as in the “Cookieless Session State Enabled” vulnerability discussed above, enabling cookieless authentication in your application can lead to session hijacking.

Vulnerable:	Secure:
<pre><configuration> <system.web> <authentication mode="Forms"> <forms cookieless="UseUri"></pre>	<pre><configuration> <system.web> <authentication mode="Forms"> <forms cookieless="UseCookies"></pre>

When a session or authentication token appears in the request URL rather than in a secure cookie, an attacker with a network monitoring tool can easily take over that session and effectively impersonate a legitimate user. However, session hijacking has far more serious consequences after a user has been authenticated. For example, online shopping sites generally allow users to browse without having to provide an ID and password. But when users are ready to make purchases, or when they want to view their orders, they have to login and be authenticated by the system. After logging in, sites provide access to more sensitive data, such as a user's order history, billing address, and credit card number. Attackers hijacking a user's session before authentication can't usually obtain much useful information. But if the

Top 10 Security Vulnerabilities in .NET Configuration Files

attacker hijacks the session *after* authentication, all that sensitive information could be compromised.

The best way to prevent session hijacking is to disable cookieless authentication and force the use of cookies for storing authentication tokens. This is done by changing the `cookieless` attribute of the `<forms>` element to the value `UseCookies`.

7. Failure to Require SSL for Authentication Cookies

Web applications use the Secure Sockets Layer (SSL) protocol to encrypt data passed between the web server and the client. Using SSL means that attackers using network sniffers will not be able to interpret the exchanged data. Rather than seeing plaintext requests and responses, they will see only an indecipherable jumble of meaningless characters. You can require your forms to use SSL by setting the `requireSSL` attribute of the `<forms>` element to `true`.

Vulnerable:	Secure:
<pre data-bbox="240 1423 792 1602"><configuration> <system.web> <authentication mode="Forms"> <forms requireSSL="false"></pre>	<pre data-bbox="824 1423 1377 1602"><configuration> <system.web> <authentication mode="Forms"> <forms requireSSL="true"></pre>

Top 10 Security Vulnerabilities in .NET Configuration Files

The previous section discussed the importance of transmitting the authentication token in a cookie, rather than embedding it in the request URL. However, disabling cookieless authentication is just the first step towards securing the authentication token. Unless requests made to the web server are encrypted, a network sniffer will still be able to read the authentication token from the request cookie. An attacker would still be able to hijack the user's session.

At this point, you might be wondering why it is necessary to disable cookieless authentication, since it is very inconvenient for users who won't accept cookies, and seeing as how the request still has to be sent over SSL. The answer is that the request URL is often persisted regardless of whether or not it was sent via SSL. Most major browsers save the complete URL in the browser history cache. If the history cache were to be compromised, the user's login credentials would be as well. Therefore, to truly secure the authentication token, you must require the authentication token to be stored in a cookie, and use SSL to ensure that the cookie be transmitted securely. By setting the `requireSSL` attribute of the `<forms>` element to `true`, the ASP.NET application will use a secure connection when transmitting the authentication cookie to the web server. Note that IIS requires additional configuration steps to support SSL. You can find instructions for configuring SSL for IIS on MSDN [here](#).

Top 10 Security Vulnerabilities in .NET Configuration Files

8. Sliding Expiration Used

All authenticated ASP.NET sessions have a timeout interval. The default timeout value is 30 minutes. After 30 minutes of inactivity, the user will automatically be timed out and forced to re-authenticate his credentials.

Vulnerable:	Secure:
<pre><configuration> <system.web> <authentication mode="Forms"> <forms slidingExpiration="true"></pre>	<pre><configuration> <system.web> <authentication mode="Forms"> <forms slidingExpiration="false"></pre>

The `slidingExpiration` setting is a security measure used to reduce risk in case the authentication token is stolen. When set to `false`, the specified timeout interval becomes a fixed period of time from the initial login, rather than a period of inactivity. Attackers using a stolen authentication token have, at maximum, only the specified length of time to impersonate the user before the session times out. Because typical attackers have only the token, and don't really know the user's credentials, they can't log back in as the legitimate user, so the stolen authentication token is now useless and the threat is mitigated. When sliding expiration is enabled, as long as an attacker makes at least one request to the system every 15 minutes (or half of the timeout interval), the session will remain open indefinitely. This gives attackers more opportunities to steal information and cause other mischief.

Top 10 Security Vulnerabilities in .NET Configuration Files

To avoid this altogether, you can disable sliding expiration by setting the `slidingExpiration` attribute of the `<forms>` element to `false`.

9. Non-Unique Authentication Cookie Used

Over the last few sections I hope I have successfully demonstrated the importance of storing your application's authentication token in a secure cookie value. But a cookie is more than just a value; it is a name-value pair. As strange as it seems, an improperly chosen cookie name can create a security vulnerability just as dangerous as an improperly chosen storage location.

Vulnerable:	Secure:
<pre><configuration> <system.web> <authentication mode="Forms"> <forms name=".ASPXAUTH"></pre>	<pre><configuration> <system.web> <authentication mode="Forms"> <forms name="{abcd1234...}"></pre>

The default value for the name of the authentication cookie is `.ASPXAUTH`. If you have only one web application on your server, then `.ASPXAUTH` is a perfectly secure choice for the cookie name. In fact, any choice would be secure. But, when your server runs more than one ASP.NET application, it becomes critical to assign a unique authentication cookie name to each application. If the names are not unique, then users logging into any of the

Top 10 Security Vulnerabilities in .NET Configuration Files

applications might inadvertently gain access to all of them. For example, a user logging into the online shopping site to view his order history might find that he is now able to access the administration application on the same site and change the prices of the items in his shopping cart.

The best way to ensure that each web application on your server has its own set of authorized users is to change the authentication cookie name to a unique value. Globally Unique Identifiers (GUIDs) are excellent choices since they are guaranteed to be unique. Microsoft Visual Studio helpfully includes a tool that will automatically generate a GUID for you. You can find this tool in the Tools menu with the command name "Create GUID". Copy the generated GUID into the `name` attribute of the `<forms>` element in the configuration file.

10. Hardcoded Credentials Used

Vulnerable:	Secure:
<pre> <configuration> <system.web> <authentication mode="Forms"> <forms> <credentials> ... </credentials> </forms> </authentication> </system.web> </configuration> </pre>	<pre> <configuration> <system.web> <authentication mode="Forms"> <forms> </forms> </authentication> </system.web> </configuration> </pre>

Top 10 Security Vulnerabilities in .NET Configuration Files

A fundamental difficulty of creating software is that the environment in which the application will be deployed is usually not the same environment in which it is created. In a production environment, the operating system may be different, the hardware on which the application runs may be more or less powerful, and test databases are replaced with live databases. This is an issue for creating applications that require authentication because developers and administrators often use test credentials to test the system. The question is: Where do the test credentials come from? For convenience, to avoid forcing developers from spending time on creating a credential store used solely for test purposes (and which would subsequently be discarded when the application went to production), Microsoft added a section to the `web.config` file that you can use to quickly add test users to the application. For each test user, the developer adds an element to the configuration file with the desired user ID and password as shown below:

```
<authentication mode="Forms">
  <forms>
    <credentials>
      <user name="bob" password="bob"/>
      <user name="jane" password="Elvis"/>
    </credentials>
  </forms>
</authentication>
```

Top 10 Security Vulnerabilities in .NET Configuration Files

While undeniably convenient for development purposes, this was *never intended* for use in a production environment. Storing login credentials in plaintext in a configuration file is simply not secure. Anyone with read access to the `web.config` file could access the authenticated web application. It is possible to store the SHA-1 or MD5 hash of the password value, rather than storing the password in plaintext. This is somewhat better, but it is still not a secure solution. Using this method, the user name is still not encrypted. First, providing a known user name to a potential attacker makes it easier to perform a brute force attack against the system. Second, there are many reverse-lookup databases of SHA-1 and MD5 hash values available on the Internet. If the password is simple, such as a word found in a dictionary, then it is almost guaranteed to be found in one of these hash dictionaries. The most secure way to store login credentials is to not store them in the configuration file. Remove the `<credentials>` element from your `web.config` files in production applications.

You're not Out of the Woods Yet

Now that you've finished reading the top-ten list, and you've checked your configuration settings, your applications are secure forever, right? Not just yet. `web.config` files operate in a hierarchical inheritance manner. Every `web.config` file inherits values from any `web.config` file in a parent directory. That `web.config` file in turn inherits values from any `web.config` file in *its* parent directory, and so on. All `web.config` files on the system inherit from the global configuration file called `Machine.config` located in the

Top 10 Security Vulnerabilities in .NET Configuration Files

.NET framework directory. The effect of this is that the runtime behavior of your application can be altered simply by modifying a configuration file in a higher directory.

This can sometimes have unexpected consequences. A system administrator might make a change in a configuration file in response to a problem with a completely separate application, but that change might create a security vulnerability in your application. For example, a user might report that he is not able to access the application without enabling cookies in his browser. The administrator, trying to be helpful, modifies the global configuration file to allow cookieless authentication for all applications.

To keep your application-specific settings from being unexpectedly modified, the solution is to *never rely on default setting values*. For example, debugging is disabled by default in configuration files. If you're examining the configuration file for your application and you notice that the debug attribute is blank, you might assume that debugging is disabled. But it may or may not be disabled—the applied value depends on the value in parent configuration settings on the system. The safest choice is to always explicitly set security-related values in your application's configuration.

Ultimately, securing a web application requires the efforts and diligence of many different groups, from quality assurance to security operations. However, the developer who codes the application itself has an inherent responsibility to instill security into the application from the beginning of the

Top 10 Security Vulnerabilities in .NET Configuration Files

development process. By making security-conscious decisions from the beginning, developers can create applications that users can trust with their confidential information and that are capable of withstanding attacks launched by hackers. As I've tried to show here, sometimes that process can be as simple as making the right decisions when configuring your application.

Top 10 Security Vulnerabilities in .NET Configuration Files

About SPI Labs

SPI Labs is the dedicated application security research and testing team of S.P.I. Dynamics, Inc. (www.spidynamics.com). Composed of some of the industry's top security experts, SPI Labs is specifically focused on researching security vulnerabilities at the web application layer. The SPI Labs mission is to provide objective research to the security community and give organizations concerned with their security practices a method of detecting, remediating, and preventing attacks upon the web application layer.

SPI Labs' industry leading security expertise is evidenced via continuous support of a combination of assessment methodologies which are used in tandem to produce the most accurate web application vulnerability assessments available on the market. This direct research is utilized to provide daily updates to SPI Dynamics' suite of security assessment and testing software products. These updates include new intelligent engines capable of dynamically assessing web applications for security vulnerabilities by crafting highly accurate attacks unique to each application and situation, and daily additions to the world's largest database of more than 5,000 application layer vulnerability detection signatures and agents. SPI Labs engineers comply with the standards proposed by the Internet Engineering Task Force (IETF) for responsible security vulnerability disclosure.

Information regarding SPI Labs policies and procedures for disclosure are outlined on the SPI Dynamics web site at:

<http://www.spidynamics.com/spilabs.html>.

Top 10 Security Vulnerabilities in .NET Configuration Files

About the Author

Bryan Sullivan is a development manager for SPI Dynamics (www.spidynamics.com), the leading provider of Web application security testing software and services. At SPI Dynamics, Bryan manages the DevInspect and QAInspect products, which can automatically detect security vulnerabilities during the development and QA phases of the software development lifecycle. Bryan is currently co-authoring a book on AJAX security for the publisher Addison-Wesley, which will be published in the summer of 2007.

Contact Information

S.P.I. Dynamics
115 Perimeter Center Place
Suite 1100
Atlanta, GA 30346

Telephone: (678) 781-4800
Fax: (678) 781-4850
Email: info@spidynamics.com
web: www.spidynamics.com