

Top 10 Application Security Vulnerabilities in Web.config Files – Part One

By Bryan Sullivan

These days, the biggest threat to an organization's network security comes from its public Web site and the Web-based applications found there. 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, making application security an issue. 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 to application security and are designing their Web-based applications with security in mind. By "baking in" application 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 Web.config configuration files of your application. Incorrectly configured Web-based applications can be just as dangerous as those that have been incorrectly coded. To make matters worse, many configuration settings actually *default* to insecure values.

This article lists five of the "worst offenders" of misconfigurations of application security that are universally problematic for all ASP.NET Web-based applications. Part two of this article will list an additional five misconfigurations that are specifically applicable to ASP.NET sites that use Web Forms authentication. So without further ado, let's get started!

1. Custom Errors Disabled

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

Vulnerable configuration:

```
<configuration>
  <system.web>
    <customErrors mode="Off">
```

Secure configuration:

```
<configuration>
```

```
<system.web>
```

```
<customErrors mode="RemoteOnly">
```

In itself, knowing the source of an error may not seem like a risk to application security, but 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. An error message can be a gold mine of information to an attacker. A default ASP.NET error message lists the specific versions of ASP.NET and the .NET framework which are being used by the Web server, as well as the type of exception that was thrown. Just knowing which Web-based applications are used (in this case ASP.NET) compromises application security by telling the attacker that the server is running a relatively recent version of Microsoft Windows and that Microsoft Internet Information Server (IIS) 6.0 or later is being used as the Web server. The type of exception thrown may also help the attacker to profile Web-based applications; for example, if a `SqlException` is thrown, then the attacker knows that the application is using some version of Microsoft SQL Server.

You can build up application security to prevent such information leakage by modifying the `mode` attribute of the `<customErrors>` element to `On` or `RemoteOnly`. This setting instructs Web-based applications to display a nondescript, generic error message when an unhandled exception is generated. Another way to circumvent this application security issue is to redirect the user to a new page when errors occur by setting the `defaultRedirect` attribute of the `<customErrors>` element. This approach can provide even better application security because the default generic error page 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).

2. Leaving Tracing Enabled in Web-Based Applications

The trace feature of ASP.NET is one of the most useful tools that you can use to ensure application security by debugging and profiling your Web-based applications.

Unfortunately, it is also one of the most useful tools that a hacker can use to attack your Web-based applications if it is left enabled in a production environment.

Vulnerable configuration:

```
<configuration>
```

```
<system.web>
```

```
<trace enabled="true" localOnly="false">
```

Secure configuration:

```
<configuration>
```

<system.web>

<trace enabled="false" localOnly="true">

When the <trace> element is enabled for remote users of Web-based applications (`localOnly="false"`), any user can view an incredibly detailed list of recent requests to the application simply by browsing to the page `trace.axd`. If a detailed exception message is like a gold mine to a hacker looking to circumvent application security, a trace log is like Fort Knox! A trace log presents a 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 looking for a way around application security would obviously find the form variable histories 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 Web-based applications 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 from Web-based applications is to disable the trace viewer completely by setting the `enabled` attribute of the <trace> element to `false`. If you have to 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`. That allows users to access the trace viewer only from the Web server and disables viewing it from any remote machine, increasing your application security.

3. Debugging Enabled

Deploying Web-based applications in debug mode is a very common mistake. Virtually all Web-based applications require 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, compromising application security.

Vulnerable configuration:

<configuration>

```
<system.web>
```

```
<compilation debug="true">
```

Secure configuration:

```
<configuration>
```

```
<system.web>
```

```
<compilation debug="false">
```

Like the first two application 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 Web-based applications. For example, if you have enabled debugging and disabled custom errors in your application, then any error message displayed to an end user of your Web-based applications 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.

Unfortunately, this configuration setting isn't the only way that source code might be displayed to the user. Here's a story that illustrates why developers shouldn't concentrate solely on one type of configuration setting to improve application security. In early versions of Microsoft's ASP.NET AJAX framework, some controls would return a stack trace with source code to the client browser whenever exceptions occurred. This behavior happened whenever debugging was enabled, regardless of the custom error setting in the configuration. So, even if you properly configured your Web-based applications 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 in part two of this article, it's safer to explicitly set the desired value rather than relying on the defaults to protect application security.

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 on a per-cookie basis, you also can set it globally in the site configuration.

Vulnerable configuration:

```
<configuration>
```

```
<system.web>
```

```
<httpCookies httpOnlyCookies="false">
```

Secure configuration:

```
<configuration>
```

```
<system.web>
```

```
<httpCookies httpOnlyCookies="true">
```

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 Web-based applications from Cross-Site Scripting attacks. A hacker initiates a Cross-Site Scripting (also called CSS or XSS) attack by attempting to insert his own script code into the Web page to get around any application security in place. 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 may be susceptible to an XSS attack.

Message boards, forums, and wikis are also often vulnerable to application security issues. In these sites, 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.

As mentioned earlier, it is possible to enable `HttpOnly` programmatically on any individual cookie by setting the `HttpOnly` property of the `HttpCookie` object to `true`. However, 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 configuration:

```
<configuration>  
  
  <system.web>  
  
    <sessionState cookieless="UseUri">
```

Secure configuration:

```
<configuration>  
  
  <system.web>  
  
    <sessionState cookieless="UseCookies">
```

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/(123456789ABCDEFG)/default.aspx`. In this case, `123456789ABCDEFG` represents the current user's session token. A different user browsing the site at the same time would receive a completely 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 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

These first five Web.config vulnerabilities that we've discussed in this article have been applicable to all ASP.NET Web applications regardless of their methods of authentication, or even whether they use authentication at all. Part two of this article details an additional five vulnerabilities that apply only to applications using Forms authentication. These misconfigurations can be even more dangerous than the first five, giving intruders the ability to access supposedly secure areas of your Web site. Finally, we will also discuss some methods of locking down your configuration files so that they can't be modified unintentionally.

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 12 years of experience in the information technology industry. Bryan is currently coauthoring a book with noted security expert Billy Hoffman on Ajax security, which will be published in summer 2007 by Addison-Wesley.