



MSDN Home

Developer Centers

Library

Downloads

How to Buy

Subscribers

Worldwide

Search for

MSDN® Magazine

The Microsoft Journal for Developers

[MSDN Home](#) > [MSDN Magazine](#) > [November 2005](#)

Advanced Search

MSDN Magazine Home

November 2005

Search

Source Code

Back Issue Archive

Column Archive

Next Issue

RSS

Subscribe

Order Back Issues

Reader Services

Meet the Staff

Meet the Authors

Media Kit

Special CD and DVD Offers

Submit an Article

Write to Us

Corrections

TechNet Magazine

Are You Protected?

Design and Deploy Secure Web Apps with ASP.NET 2.0 and IIS 6.0

[Michael Volodarsky](#)

Print

Parts of this article are based on a prerelease version of ASP.NET 2.0. Those sections are subject to change.

This article discusses:

- Protecting resources and access control
- Authentication and authorization
- Code access security in ASP.NET
- Application auditing

This article uses the following technologies:
ASP.NET, IIS 6.0, SQL Server

[+ Contents](#)

Web applications are among the most common computing services that are exposed to the Internet, and thus they pose an inviting target to anyone who wants to break into your network to steal sensitive information, tamper with your data, or otherwise compromise your system. Ensuring the security of a Web application is a serious task, and requires consideration throughout the design, development, deployment, and operation phases. It should not be viewed as something that can be slapped onto an existing application, or achieved simply by applying existing platform security features.

Even when developed for use within a relatively secure platform, a Web application must follow the security best practices of the platform throughout the design, development, and deployment stages to provide the maximum level of protection against attack. In order to produce secure Web app, this platform-specific knowledge should be combined with secure design practices, threat modeling analysis, and security penetration testing. This article discusses best practices that allow you to take advantage of the security features of ASP.NET 2.0 and IIS 6.0 to build and deploy more secure Web applications.

Designing Secure Web Applications

Secure design principles usually boil down to a few key principles: assume all input into the system is malicious, reduce the exposed surface area of the system, lock down the system by default, and use defense-in-depth rather than relying on other parts of the system for protection. Following these general principles from the design phase onward is key to making sure your application is as well protected as possible.

Threat modeling analysis is used to map out the data flow inside a system and to examine the possible entry points a malicious user may be able to exploit for access to

**ORDER
TODAY!**

**Visual Basic
2005 BONUS
information
ORDER YOUR
DVD TODAY!**

the system. Threat modeling is a key exercise for changing your perspective from that of a designer to that of an attacker, helping you to discover the potential security holes in your application. For more information on threat modeling, see [Threat Modeling](#). Security penetration testing is another name for hacking your own application—attempting to directly compromise your application to find problems before others do it for you. You can try pushing the application to its limits by sending invalid or worst-case data. You can also try foiling your application by using your knowledge of its internal workings—in this case, your insider knowledge gives you an unfair advantage over typical attackers, but may also unearth vulnerabilities that were otherwise buried deep within your code. There are a variety of tools available that can help you do penetration testing.

[Back to Contents](#) 

Protecting Resources

Usually, the first task of deploying a Web application is making sure that it does not expose sensitive resources that can be accessed by anonymous clients on the Internet. If this is an intranet application, this is usually accomplished by making sure that all clients are authenticated with Windows® authentication (more on access control later), and that the application is protected from external access by a firewall. For an Internet application, this is more of an issue because it almost always needs to be at least minimally accessible by anonymous Internet users. Aside from this difference, the following guidance is nonetheless equally applicable to protecting resources for both Internet and intranet applications.

Ideally, it is best to make sure that your application's Web namespace does not contain any files that are not intended to be served to the client. This means removing all such files from the physical directory structure starting with the topmost directory marked as a Web application or virtual directory in the IIS configuration. If the file is not in the Web namespace, it should not be accessible by making requests to that namespace, unless your application code takes direct steps to open the file and serve its contents. If your application is programmatically accessing data or supporting files during its execution, you should place them outside the Web namespace.

If you cannot remove all such files, make sure that IIS is configured not to serve these files to Web clients. This should be accomplished by mapping the extensions for the protected files to ASP.NET in the IIS script processor map configuration (see **Figure 1**), and subsequently mapping those extensions to the `HttpForbiddenHandler` in the ASP.NET `<httpHandlers>` configuration for the application or directory. By default, ASP.NET already blocks access to a number of extensions common to Web applications, including `.cs`, `.java`, `.mdb`, `.mdf`, `.vb`, and others.

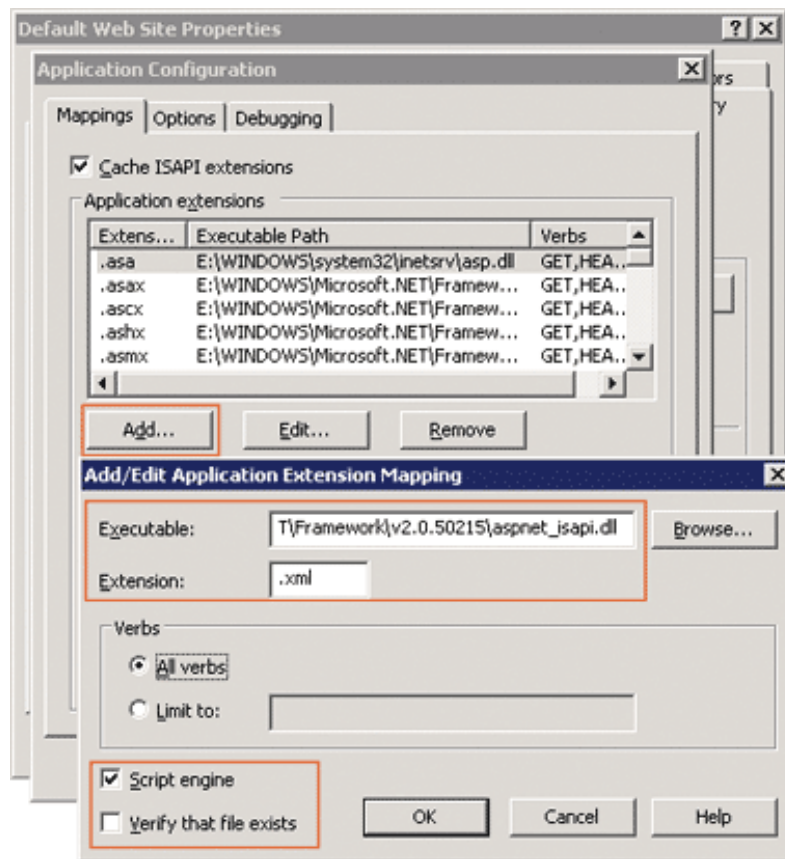


Figure 1 Mapping XML Files to the ASP.NET ISAPI Extension

Configuring the Web.config as follows prohibits the .xml extension from being served if the extension has been mapped to ASP.NET for this application:

```
<configuration>
  <system.web>
    <httpHandlers>
      <add path="*.xml" verb="*"
          type="System.Web.HttpForbiddenHandler" />
    </httpHandlers>
  </system.web>
</configuration>
```

Note that this configuration generates a "403 Forbidden" response with an ASP.NET error message that gives away the existence of this file. You can instead use the System.Web.HttpNotFoundHandler in order to mask the existence of the file and return a generic 404 response.

If you uninstall ASP.NET, the IIS script processor maps for it will be removed and the unmapped extensions will be processed by the IIS static file handler. It is frequently thought that removing a corresponding extension entry from the IIS MIME mapping configuration will prevent the static file handler from serving files with that extension. In actuality this is not always true because the static file handler will also honor the registry MIME mapping configuration and therefore will still serve the extension if the registry entry is present. Because of this, you would have to make sure that both the metabase MIME mapping configuration and the registry key in HKEY_CLASSES_ROOT do not contain an extension entry for the forbidden extension. Due to the management complexity, this practice is not recommended as a way of securing content. If you do use

this method, you should also make the files hidden for additional protection because the IIS static file handler will not serve files that have been given the hidden attribute.

Of course, this guidance regarding the IIS static file handler only applies to extensions that are not mapped to an ISAPI extension in the IIS script mapping configuration. If the extension is mapped, then the ISAPI extension in the corresponding IIS script mapping will be responsible for controlling access to requests that have that extension.

You can also take advantage of the protected directories feature in ASP.NET 2.0. By default, ASP.NET 2.0 will block access to all URLs that contain directory segments named App_Data, App_Code, App_Browsers, App_WebReferences, App_GlobalResources, and App_LocalResources anywhere in the URL. ASP.NET 1.1 and 2.0 will both also block access to the /Bin directory. This support is conditional on having the aspnet_filter.dll ISAPI filter registered with IIS, which is done by default during ASP.NET installation. By placing your content in these directories, you can prevent HTTP access to it regardless of the file extensions being requested. [Figure 2](#) describes the protected directories under ASP.NET 2.0. URLs containing other App_* segments, including the App_Themes directory, are not blocked by ASP.NET.

[Back to Contents](#) 

Access Control

Most Web applications provide multiple levels of access to data and resources, and need to ensure that clients receive appropriate access based on their credentials. Access control in Web applications is typically handled in two stages: authentication and authorization. Authentication refers to the process of determining the identity of the client making the request. Authorization refers to the process of determining whether that client identity is allowed to access a particular resource. **Figure 3** illustrates the security-relevant stages of request processing of a typical HTTP request on the IIS/ASP.NET platform.

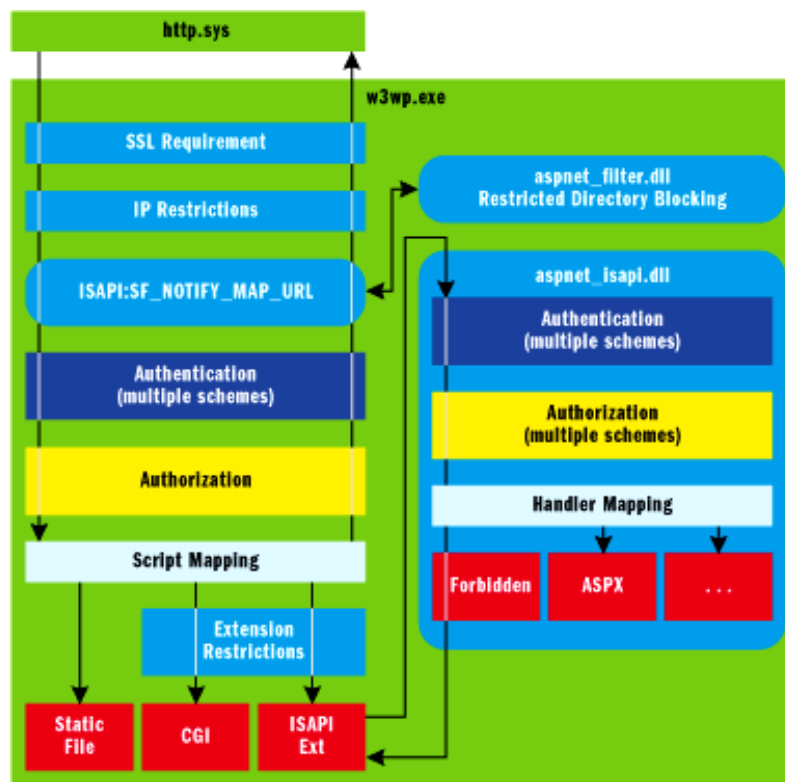


Figure 3 Processing an HTTP Request

There are a number of non-user access control mechanisms available that you may want to consider for your application to reduce its attack surface. These mechanisms on IIS 6.0 include the IP restriction list and the Web service extension restriction list.

The IP restriction list allows you to specify the client IP addresses that are allowed access into your application. You can configure specific IP addresses and domains as well as subnets. Consider doing this if your application is intranet only, or if your clients always access your application from a specific IP address or domain. When applicable, this should be used as a defense-in-depth measure in conjunction with other access control methods.

The extension restriction list allows you to globally define the ISAPI extension DLL files and the CGI .exe files that can be executed on the server. By default, all such extensions are prohibited, and you should enable only the ones that you need and leave the other ones disabled. Third-party product installations may enable additional components, so you might need to disable them if you are not using them to serve Web content.

Previous versions of IIS also benefited from the UriScan ISAPI filter, which provided additional request validation and blocking of dangerous input. IIS 6.0 has been reengineered to provide greater security than previous versions, and no longer exposes vulnerabilities that required UriScan for protection. However, if your security requirements are not covered by the features provided in IIS 6.0 by default, you may want to continue using UriScan. The information available at [Determining Whether to Use UriScan 2.5 with IIS 6.0](#) can help you decide whether to use UriScan.

[Back to Contents](#) 

Authentication

IIS and ASP.NET together provide a foundation for building access control mechanisms,

and ASP.NET 2.0 further expands upon this to provide ready-to-go application blocks that can be used to deploy such mechanisms quickly.

The output of IIS authentication mechanisms is always a Windows identity that represents the user from whom the request is being made. IIS provides built-in support for the following authentication types: anonymous, Integrated Windows, Basic, Digest, certificate mapping, and Microsoft® Passport. [Figure 4](#) illustrates when different authentication modes are typically used.

Intranet applications typically will use the Integrated Windows authentication to authenticate clients with their domain accounts; alternatively, they may use certificate authentication, though certificates are more common in the Internet realm. Internet applications should use Basic authentication over SSL or Digest authentication if users have Windows accounts, or anonymous authentication with an application-specific authentication scheme if they do not have Windows accounts.

If anonymous authentication is not enabled for a URL, then IIS will either make sure that the request is authenticated or it will reject the request. This should be used as a defense-in-depth measure for applications using other IIS authentication mechanisms (most intranet applications) to outright reject all unauthenticated users if no anonymous access is desired.

ASP.NET supports three types of authentication: Windows, Forms, and Passport. Windows authentication simply accepts the Windows identity produced during IIS authentication. Forms authentication implements a ticket-based application layer authentication scheme that should be used by ASP.NET applications that do not associate Windows accounts with their users. (Forms authentication should be combined with the IIS anonymous authentication option.) Passport authentication uses the Microsoft Passport authentication scheme. The authentication scheme for each application is specified in the <system.web><authentication> configuration element, as shown here:

```
<configuration>
  <system.web>
    <authentication mode="Forms" />
  </system.web>
</configuration>
```

With forms authentication, when the user is denied access for a resource that requires authentication, the user is then redirected to a configured URL that points to a login page. The login page is a custom ASP.NET page written by the application developer, which provides a way for the user to supply credentials. The login page then validates the credentials against a custom user store, such as a SQL database, and uses the `System.Web.FormsAuthentication` class to issue an encrypted authentication ticket to the client. This ticket can be either a cookie or a special blob that augments the URL (starting with the ASP.NET 1.1 Mobile Toolkit and ASP.NET 2.0). On every subsequent request, the client presents this ticket, and the forms authentication mechanism automatically processes it to generate the user identity for the request.

With forms authentication, the complexity of issuing and managing the authentication ticket is absorbed by the ASP.NET framework. However, you still have to write the login

page and the associated user store implementation. ASP.NET 2.0 simplifies the entire process by providing two additional components: the Membership service and the Login control family.

The Membership service, exposed by the `System.Web.Security.Membership` class, provides a useful abstraction for creating, managing, and validating user credentials. Like many other ASP.NET 2.0 features, the provider-based design enables the service to work with a variety of back-end data stores, including SQL Server™ and Active Directory® providers that are included in the ASP.NET framework. Setting up your own user database implementation is as easy as deploying the Membership database schema to your SQL Server or SQL Server Express instance through the `aspnet_regsql.exe` tool, and configuring a `SqlMembershipProvider`—or just using the default provider that automatically creates the SQL Server Express membership database in your application's `App_Data` directory on first access.

The Membership service provides support for many common user management functions, including password hashing, account lockout, and login tracking. You can learn more about the Membership service in the ASP.NET 2.0 Quickstart ([Using the Membership and Role Manager APIs](#)), and in Dino Esposito and Andrea Saltarello's article in this issue of *MSDN® Magazine*.

With the ASP.NET 2.0 Login control, the crux of the login page creation is simplified even further by providing a drag and drop control that implements all of the login page functionality automatically by working against membership and forms authentication APIs to validate the user credentials and create the authentication ticket on successful logon. The Login control provides flexible functionality, including working with custom Membership providers, performing question/answer password reset, and doing custom credential validation. A set of supporting controls, including `LoginView` and `CreateUser` controls, can be used to support the rest of your user management interface (see [Using the Login Controls](#)).

When using custom application authentication with forms authentication or your own authentication mechanism, protect your login page and preferably your entire application with the secure sockets layer (SSL) to prevent disclosure of login credentials and authentication tickets. (I'll cover authentication ticket security in more detail later in this article.) Also, you should require your users to use strong passwords. The Membership service provides the ability to set regular expressions for enforcing password strength and format. You should make sure that hashed passwords are used (which is the default behavior), and you should not store credentials in the forms authentication `<credentials>` section in your application configuration file.

[Back to Contents](#) 

Authorization

After a user identity for the request has been established, the identity is stored in the request context as a `System.Security.Principal.IPrincipal` object, which is available from `HttpContext.Current.User`. This object can then be used to make access control decisions for the request.

ASP.NET provides two built-in authorization mechanisms that control access on the URL level: File authorization and URL authorization. Both of these execute after authentication, in the AuthorizeRequest ASP.NET pipeline stage, and check whether the requesting identity has access to the requested URL. If the identity is determined not to have access, the request is terminated with a "401 Unauthorized" response.

File authorization works only when the Windows authentication option is used (when the user principal contains a Windows identity), and uses file access control lists (ACLs) to determine whether the requesting identity should be allowed access to the URL. Note that File authorization also only works when the the request can be mapped to a file and the file is on a local machine. If the physical file is on a share, this authorization has no effect and allows the request.

File authorization can be used by applications that authenticate users who have Windows accounts and who use ACLs to control access to their resources. Be sure to remove read access for any user identity that should not have access to a given file when using this type of authorization.

URL authorization uses configuration-based access control rules that reference the user's name or roles as a way to grant or deny access. It is commonly the preferred authorization mechanism because it does not depend on the user identity being a Windows account, it can be used with any authentication mechanism that produces a user principal, and it offers an easier way to manage access policies than by managing ACLs on files.

URL authorization access control rules can be specified per-URL in the <system.web><authorization> configuration section. The authorization mechanism will process the rules in the top-down order and select the first rule that is matched, taking the deny or allow action specified by the rule. The following configuration allows all authenticated users and denies anonymous users access for the URL scope:

```
<authorization>
  <allow users="*" />
  <deny users="?" />
</authorization>
```

Alternatively, you can also qualify the rules with user names or roles. In ASP.NET 1.1, you have to manually associate a list of roles with a user principal after authentication. In ASP.NET 2.0, you can use the System.Web.Security.RoleManager class to create and manage roles and have these roles automatically associated with a user after authentication. You can also use location tags to specify access control rules for specific URLs in the same Web.config file.

URL authorization should be used only to allow access to the specific set of users, and to deny access to everyone else. A good practice involves designing a more restrictive configuration at the root directory level (deny everyone or deny all anonymous) with specific relaxation for individual URLs or subfolders that have less stringent access requirements. You should consider the authorization requirements of your application early in the design stage, and try to group URLs with similar access requirements together in directories with a single set of authorization rules to minimize the need to

configure many individual rules, and to reduce the risk of getting them wrong.

[Back to Contents](#) 

Granular Application Authorization

URL-level authorization is a great way to secure your application, and should be used wherever possible to restrict access to the minimum set of users that need to interact with your system. However, sometimes further application-level authorization is required in order to provide more fine-grained control over the functionality that is available to your users.

Typically, application authorization is performed by inspecting the current authenticated user principal, available through the `HttpContext.Current.User` property, and making a decision to grant or deny access to the specific functionality based on either the user name or role membership of the principal. In ASP.NET applications, this is typically done in two stages: hiding page regions, and validating access programmatically before performing server actions.

Hiding page regions is a common practice in existing applications, and entails selectively showing a set of page controls by placing these controls in a panel that is set to be visible or invisible based on the user identity or role membership. In ASP.NET 2.0, this task is simplified via the `LoginView` control, which provides an easy way to associate templates with users and roles. (Take a look at [Using the Login Controls](#) for an example on how to use this control.) Note, however, that this practice alone is not enough.

In addition to hiding portions of a page from being visible, you need to enforce access control in the actual actions taken by your application in response to user input. These actions are most commonly ASP.NET page postbacks that invoke server event handlers, or code in your page event handlers that acts on a querystring, form data, or other client input. In ASP.NET 1.1, hiding a control does not prevent a malicious client from being able to trigger its event handlers, so this is a mandatory practice. In ASP.NET 2.0, the controls in the `System.Web.UI.WebControls` and `System.Web.UI.HtmlControls` namespaces prevent by default postback events from being raised if they are hidden, and custom controls can also do the same.

To enforce access control, place code in every event handler or method that performs a sensitive action to make sure the current user is authorized to perform the current action. This way, even if the client manages to directly invoke the event handler, you will ensure that only authorized users can perform the protected action. You can do this by directly checking the user principal, or by using declarative or imperative demands for the user principal.

The code shown in [Figure 5](#) ensures that the `CreateDatabase` event handler only allows the Administrator user name or all users in the Operator role to complete the call. Alternatively, you can enforce code authorization by making demands against the `System.Security.Permissions.PrincipalPermission` class. This class can be used to make demands for a particular user name and role against the current user principal set on the thread. This can be used to do both programmatic and declarative demands against the user principal. [Figure 6](#) shows two examples of using the `PrincipalPermission` to require

the user to be Administrator.

ASP.NET will set the current request principal on the thread after all authentication modules and the global.asax AuthenticateRequest event handlers have executed. This means that if you want to set a custom principal on the request after this stage, you have to set the thread principal manually by assigning your principal object to the Thread.CurrentPrincipal property.

[Back to Contents](#) 

Resource Access

In addition to the user-based authentication and authorization I've discussed, it is also important to consider how Windows resources (files, registry, objects, remote SQL Servers) are accessed within an ASP.NET application. Because ASP.NET user abstraction is decoupled from Windows accounts, the identity used for accessing Windows resources is not always the identity of the authenticated user. In fact, the accessing identity depends on the Windows identity impersonated by an application thread, and can be any of the identities described in [Figure 7](#).

Intranet applications may choose to impersonate the client in order to limit the application to the rights of the accessing user. However, this also increases the burden of managing the resource permissions (file ACLs and SQL Server permissions, for example) for the application, and introduces performance overhead due to, among other reasons, inefficient database connection pooling. Additional limitations exist for remote resource access, such as accessing a remote SQL Server database with Integrated Authentication, because most IIS authentication schemes do not produce authentication tokens that can access remote machines on your network (except for Basic authentication, or if you have configured Windows NT® Lan Manager or Kerberos delegation).

An alternative is to use the trusted subsystem design, where your Web application accesses internal resources with a single identity, and ensures that its users have the permission to access these resources by doing application-level authorization (described previously). This is the approach taken by default by ASP.NET with the no-impersonation process identity mode. You can also achieve the same result with a fixed application identity, in which case you should be sure to protect the configuration credentials via configuration encryption.

When using the trusted subsystem approach, keep in mind the following best practices:

- Perform early authorization (as described earlier in this article) and deny access to all except for the authorized users.
- Run your application with the least privilege possible. If it's compromised, the attacker can only do as much damage as the process identity or application identity allows.
- Only grant the minimal necessary access to resources. With SQL Server, for example, do not grant the application identity DataBase Owner (DBO) rights. Instead, only grant rights to the stored procedures necessary for its functionality, and withhold all other rights, including SELECTs on the tables.

[Back to Contents](#) 

Client Ticket Security

ASP.NET Web applications commonly use client tickets to associate state with a particular user. For example, the ASP.NET forms authentication mechanism described previously uses an encrypted client cookie to enable a client browser to access the application for a period of time without requiring the credentials to be provided by the user on every request. Other users of client tickets include Role Manager, which uses a cookie to associate a set of application roles with the client, and Session State, which uses a cookie to identify the session id for the client session. All of these features can also use URL-based tickets in ASP.NET 2.0, as I'll describe later. Application code can also store custom state in ViewState, form fields, and custom cookies.

Because the tickets are exposed to the client, they can be stolen, replayed, and altered by malicious clients, and that can result in a number of threats against the application. Specifically, these threats include malicious clients stealing a ticket from a legitimate user, and masquerading as that client by replaying that ticket or examining it for useful information about your application. Malicious clients could also craft a custom ticket to pretend to be another user or exploit the application in a specific way.

The following is general guidance that applies to all client tickets used by an application that can help mitigate or reduce the risk of all these threats. If the ticket contains secure information that you don't want the client to know, encrypt the ticket. This is supported by the forms authentication, Role Manager, and ViewState features. For forms authentication, set the `<system.web><authentication><forms>` protection attribute to All, which is the default setting. For Role Manager, set the `<system.web><roleManager>` cookieProtection attribute to All.

For ViewState, set the `viewStateEncryptionMode` attribute to Always in the `@ Page` directive or in the `<system.web><pages>` element. Note, however, that this functionality is available in ASP.NET 2.0 only. In ASP.NET 1.1, encryption can be achieved by using the 3DES ViewState validation.

The encryption performed depends on the settings in the `<machineKey>` element. Using encryption addresses the second threat, malicious users examining the ticket for useful information about your application. The tradeoff here, of course, is performance due to the runtime overhead introduced by decrypting the ticket on each request.

At the very least, you should prevent the ticket from being modified or created on the client by enabling Message Authentication Code (MAC) validation checks. This check results in a hash signature of the ticket contents appended to the ticket, which allows the server to make sure that nobody interfered with the ticket by recomputing the hash and validating it against the signature.

If you turn on encryption for the aforementioned features, you will automatically get the benefits of MAC validation. In addition, ViewState provides the ability to inject a unique value used during validation to make sure that the view state blob was generated for the current user, which prevents other users from being able to submit a form on another user's behalf. This is accomplished by programmatically setting the `ViewStateUserKey` property to a user-unique value, such as a session ID or authenticated

user name, on every request.

All of these features allow you to downgrade the protection from encryption to validation only, which does not protect the data from being discovered, but does protect it from being changed on the client. This addresses the third threat, a malicious user crafting a custom ticket to impersonate another user.

If you have custom data you would like to store in a way that prevents it from being discovered or modified by the client, you have a few options. You can store it in the session, in which case the state is stored on the server and the client only gets the session ID. You can also store the data in the userData region of the encrypted forms authentication ticket, in ViewState, or in a custom cookie with your own MAC validation and encryption mechanism, although this requires you to manage your own keys.

With both validation and encryption enabled, you still do not address the ticket replay threat, which results from malicious clients being able to steal a valid ticket and later present it to the server, masquerading as the original client to which the ticket was issued. Another variation of this threat is a malicious client forcing the legitimate client to use the ticket provided by the malicious user. There are two general strategies that can help you reduce this risk: preventing the tickets from being stolen, and preventing tickets from being replayed.

To keep tickets from being stolen in the first place, the application should prevent the ticket from being sniffed from the network path to the client, and also ensure that the ticket is not stolen at the client. Protecting the network path is best done by enabling SSL for the entire Web site domain, or at least for the scope of authenticated URLs and the login page. If you do the latter, make sure that the path for the forms authentication, Role Manager, or custom cookie-based ticket is set to the SSL-protected segment of your URL namespace. Setting the path is achieved by either setting the path and cookiePath attributes for forms authentication and Role Manager in their respective configuration sections, or programmatically by setting the Path attribute on the System.Web.HttpCookie object that represents the cookie issued by your application. You can also intercept cookies set by other components by examining the HttpResponseMessage.Cookies collection, and forcefully set the path on those cookies before the response goes out to the client. Note that if you are using URL-based tickets, there is no sure-fire way to ask the client to restrict these tickets to a particular URL namespace. The least you can do there is avoid generating the links to URLs not covered by SSL in your application content.

Keep in mind that SSL introduces a performance penalty, so the coverage decision will depend on the requirements of your application. Using SSL this way ensures that the ticket is not sent in clear text on any portion of the application that can receive the ticket cookie (which is issued for the entire domain by default) or can be linked to with the URL-based, ViewState, or form-stored state. To learn how to enable SSL in IIS, see the Knowledge Base article [How To Enable SSL for All Customers Who Interact with Your Web Site in Internet Information Services](#).

In ASP.NET 2.0, forms authentication and Role Manager features also support the requireSSL and cookieRequireSSL configuration options, respectively, which allow you to

mandate SSL usage for all URLs that can generate and accept tickets for these features.

If you are using SSL, set these options to true to prevent accidental disclosure of the tickets when they are issued or returned over clear text connections.

Unfortunately, even if you use SSL, the tickets may be stolen on the client. This can happen on public machines, when the browser windows are left open or URLs containing URL-based tickets are bookmarked. It can also happen due to cross-site scripting attacks that can steal cookies (more on these later).

Securing public computers should begin by encouraging your users to close their browser windows when they are finished with the site, and making the Log Out button highly visible and readily available. In addition, you can consider implementing a custom JavaScript solution to detect an idle client session and pop up a window asking whether the user is still there (and if so, making a request to the server to reset the timeout timer). This practice works well with shorter server ticket timeouts. A more extreme solution is available to the forms authentication ticket, allowing you to disable the renewal of the ticket and force the client to log in again after a certain period of time regardless of their activity. This prevents the malicious client who managed to capture the ticket from using it indefinitely by renewing it, because the ticket will always expire in a fixed amount of time. This is done by setting the `slidingExpiration` attribute in the `<system.web><authentication><forms>` element to false.

In ASP.NET 2.0, cookie ticket-based features will automatically issue the cookie with the `HttpOnly` attribute set, which prevents some browsers, including Internet Explorer, from making the cookie available to JavaScript. This effectively prevents cross-site scripting attacks aimed at stealing these cookies from clients using these browsers.

Another best practice is to avoid using URL-based tickets unless you absolutely require them (for example, if you have mobile users or a requirement to support clients that disable cookies). This is because URL-based tickets are more susceptible to being discovered (bookmarked or e-mailed URLs) and to being used in a "phishing" attack (sent by e-mail or posted on a message board) to make another user share a ticket with a malicious client. The best thing to do is use cookie-based tickets for all your features, but if you cannot do that, use one of the auto-detection settings available in ASP.NET 2.0 for all ticket-based features, including forms authentication, Role Manager, and Session State. The two modes available are `UseDeviceProfile` (the recommended approach), which uses the client browser capabilities to determine whether cookies are supported, and `AutoDetect`, which performs a redirect-based auto-detection of cookie support before determining the mode. This allows you to scope the URL-based ticket usage only to clients that require it.

[Back to Contents](#) 

Preventing Replay

A few strategies are available to prevent replay of tickets. ASP.NET 2.0 session state provides an added level of protection for URL-based tickets by allowing you to reject and regenerate the session ID ticket when a corresponding session is not found on the server. This prevents multiple clients from sharing the same session ID if they happen to click on the same URL, which may have been indexed by a search engine or posted by a

malicious client. This feature is enabled by default and can be set in the `regenerateExpiredSessionId` attribute of the `<sessionstate>` configuration element. You can also attempt to validate that the client providing the ticket is the original client to whom the server issued the ticket. This practice requires you to capture unique attributes of the client and encode them into the ticket (with encryption and hash validation). Then, on each request, you can compare the attributes of the client with the encoded attributes in the ticket, rejecting the ticket if they don't match. Unfortunately, the best you can do with standard HTTP clients is the user agent request header and the client IP address. Both of these are non-unique, especially the IP address if the client is coming from behind a proxy server, used by a number of access providers including AOL, but may provide some additional level of protection. For an example on how to do this, see Jeff Prosise's *MSDN Magazine* column at [Wicked Code: Foiling Session Hijacking Attempts](#).

[Back to Contents](#) 

Preventing Attacks Through Input Validation

If you take away anything from this article, it should be the age-old security principle that all input is evil! This is especially true with Web applications, most of which receive input from remote anonymous clients. In designing your application, keep in mind that you cannot expect any level of security from the client browser. Just imagine me opening a telnet session and typing in the HTTP request manually. The client can send you any URL, headers, cookies, and form fields completely independent of how you render your pages and your JavaScript.

Because of this, keep in mind that any data your application receives from the Cookies, Headers, QueryString, and Form fields collections may not be safe. You should always validate all such input before making any sensitive decisions based on it. Keep in mind that even if your application does not appear to make any security-sensitive decisions based on the input, the components it uses might. Also, if you are storing this input in a trusted persistence medium, such as a file or database, it may be used to compromise another software system that consumes that data later.

Input validation is best done by defining what correct input to your application looks like, and rejecting all other input that does not meet this definition. This may include type validation, string length validation, type-specific range validation, and regular expression format validation. The stricter your validation, the better.

For input coming from ASP.NET form controls, you can use associated ASP.NET Validator controls to perform a lot of this validation (just make sure to always enforce server-side validation by calling `Page.IsValid`). For all other data, perform validation in your page callbacks. A good rule of thumb is to perform the validation and reject invalid input as early as possible, but you must still validate input right before it's used.

Some input data can take many forms that mean the same thing, such as Windows user names (`user@domain.com` or `domain\user`) or, even more importantly, file paths and URLs. When dealing with such data, it is critical that it is converted to a canonical format before any authorization or resource access decisions are made based on it. This is

known as canonicalization, and is frequently the source of many security vulnerabilities in software. Problems occur when two separate software layers, typically the authorization and resource access layers, make different interpretations of a non-canonical string.

To make sure that your application does not fall victim to this problem, it is best not to accept such data from the client, and instead manage it internally in your application. If you do accept such input, you need to convert input to a standard representation, and make sure that you use a standard set of APIs to process it across the different layers of your application. For example, if you are dealing with file system paths, always use the `System.IO.Path.GetFullPath` APIs to obtain a canonical representation of the path, and use other APIs on that class to work with the resulting path. (Note that native IO APIs and .NET IO APIs use a different canonicalization format, so passing .NET path strings to native code may lead to canonicalization vulnerabilities.) Use the ASP.NET 2.0 `System.Web.VirtualPathUtility` class to work with ASP.NET virtual paths. And avoid doing your own string manipulation of path data because it is quite possible that you could miss a special case and do it wrong.

The two most common input threats to a Web application are cross-site scripting and SQL injection. Cross-site scripting, also known as XSS, refers to forcing a browser to run client-side code by submitting an encoded script fragment as input to a Web application that echoes it back in a response, not necessarily to the same client. XSS can be used to steal authentication cookies, or to modify the content displayed by the browser with malicious intent. XSS attacks are mitigated by rejecting any input that may contain XSS scripts as part of your input validation strategy, and HTML-encoding the input that gets echoed back to the client to render the XSS scripts completely harmless.

ASP.NET 1.1 provides the `validateRequest` option that detects common XSS signatures in input to ASP.NET pages and terminates the request. If you can, keep this option enabled on your pages, but keep in mind that it does not apply to non-page content, and this option alone is not sufficient to provide input validation for your application.

If you are writing the response manually or developing a custom control, always HTML-encode the output of your app that is determined to contain client input by using the `HttpUtility.HtmlEncode` and `HttpUtility.UrlEncode` methods. ASP.NET 2.0 `GridView` and `DetailsView` controls provide the ability to HTML-encode parts of their output that is data bound. However, in all other cases you are responsible for HTML-encoding the strings considered unsafe before setting properties that affect control output.

As a defense-in-depth measure, some ASP.NET 2.0 features set the `HttpOnly` attribute on its cookies to prevent them from being accessible by client-side script. You should also set this property on your custom cookies.

SQL injection refers to another form of script injection, only this time through injecting specially escaped pieces of SQL queries into input parameters that are used by an application to construct dynamic SQL queries. This can be a very dangerous attack, as it can allow the attacker to steal sensitive data in your database and in some cases even modify the data.

To prevent SQL injection attacks, in addition to using strong input validation, you must

not build dynamic SQL queries based on client input. Instead, use fixed queries or stored procedures with strongly typed parameters that are initialized with client input. For an example on how to do this, see [Data Security: Stop SQL Injection Attacks Before They Stop You](#). And for a hard look at how SQL injection attacks can lead to major security vulnerabilities, see Jesper Johansson's *TechNet Magazine* article at [Anatomy of a Hack: How A Criminal Might Infiltrate Your Network](#).

[Back to Contents](#) 

ASP.NET and CAS

ASP.NET provides a code access security (CAS) model that builds on the code access security mechanisms in the .NET Framework. If you are not familiar with the .NET CAS model, see [Code Access Security](#) and the article by [Mike Downen](#) in this issue of *MSDN Magazine*. ASP.NET primarily uses its CAS model to sandbox Web application code, both as a least-privilege mechanism and also to isolate multiple ASP.NET applications on a shared server.

ASP.NET CAS is based on trust levels (see [Figure 8](#)), which can be configured for each application by setting the `trustLevel` attribute in the `<system.web><trust>` configuration section. The available trust levels are defined in the `<system.web><securityPolicy>` configuration section, and map each trust level entry to a policy file that contains the corresponding permissions given to the application in that level (the files are located in the .NET Framework version\CONFIG directory). Both sections should be locked in the machine-level configuration when the application is untrusted to prevent the application from being able to override the trust-level setting in its `Web.config` file. The exact list of permissions that are available in ASP.NET 1.1 trust files can be found in the Trust Levels section of the patterns & practices article at [Chapter 9 – Using Code Access Security with ASP.NET](#).

At run time, the set of permissions in the trust policy file is associated with all user code, including compiled assemblies in the `/Bin` directory, custom page and control code, and code in the `App_Code` directory. To be precise, the resulting set of permissions is an intersection of the application-level permissions specified in the ASP.NET trust policy file with the permissions in the enterprise, machine, and user levels of the CAS configuration hierarchy, where each lower level can only reduce the set of permissions granted by the level above.

By using trust levels with more restrictive permission sets, you can lower the damage potential of the application if it is compromised, and you can sandbox the application by restricting it to operations that cannot affect other applications on the computer. It is recommended that you test and run your applications in the medium trust level if they are trusted, and low trust level if they are not trusted (especially on a shared server). Be mindful that running in partial trust does introduce a performance penalty so it may not be appropriate in some scenarios. By default, ASP.NET uses the full trust level, which allows the application to have unrestricted access on the machine subject only to the limitations of Windows security.

Your application may encounter several common problems due to insufficient

permissions when running in partial-trust. A number of system assemblies require the caller to be fully trusted, by omitting the Allow Partially Trusted Callers Assembly (APTCA) attribute, which effectively makes them not usable in partial trust. In this situation, you have the option of either removing the usage of these assemblies from your application or encapsulating the required functionality of these assemblies in a wrapper assembly that is registered in the Global Assembly Cache (GAC) and has the APTCA attribute set, and asserting the unrestricted permission in the wrapper. If you choose to do the latter, you must be extremely careful to validate and restrict the usage of the wrapped assembly to the minimal required level, and demand another set of permissions to reduce the surface area of the new assembly.

Other code used by your application may demand permissions at run time that are not granted by the active policy file. In this situation, you still have the previous option, but you can also add the minimal level of the needed permission to the desired policy file.

This will likely be required for any code that demands custom permissions that the default ASP.NET partial-trust files do not know about. For more information on pursuing either of these options, see [Chapter 9 – Using Code Access Security with ASP.NET](#).

ASP.NET trust levels do not apply to internal ASP.NET code, which always executes with unrestricted permissions. This means that your application can potentially be configured to perform actions that are prohibited for custom code, primarily through configuration settings or declarative settings in ASP.NET pages and controls. For example, even though you cannot connect to SQL Server from your own code, you can configure the SqlDataSource control to connect and execute queries because it is part of the System.Web assembly and is registered in the GAC. ASP.NET 2.0 mitigates this problem by providing an option to execute the entire page processing code path in the application trust, including the internal page processing code path and the associated ASP.NET control processing. This is configured with the processRequestInApplicationTrust attribute in the <system.web><trust> configuration section, and is enabled by default. This setting may cause some of your code to break when it is registered in the GAC due to the assumption that it will execute with unrestricted permissions.

To deal with this issue, and also to simplify security management of permissions, a number of ASP.NET features assert all permissions required for them to execute and instead demand that the app be running in a particular trust-level, as expressed by the AspNetHostingPermission granted in each of the trust-level policy files. For example, the session state feature can be used in all trust levels, but the SQL and State Server out-of-process state storage modes require medium trust and assert all other permissions (such as SqlClientPermission and SocketPermission, respectively).

Note that CAS is completely independent from the Windows security model. CAS uses permissions granted to a piece of code, and Windows uses privileges and object ACLs expressed in terms of the Windows identity of the process or thread. CAS serves to further restrict the set of actions that managed code in a Windows process can perform, but it does not supersede Windows security or allow managed code to bypass it. CAS can only control access to managed code APIs. If an application can call native code, it can bypass all CAS restrictions. Because of this, you should always use a least-privileged

Windows identity for your worker processes.

[Back to Contents](#) 

Auditing with Web Events

Auditing is the process of monitoring application usage and collecting security-relevant information. It is an important part of application operations because it allows you to detect and stop potential security breaches, as well as investigate them after the fact so you can prevent them in the future. In some cases, auditing can help you to identify and take action against the attacker.

In ASP.NET 2.0, the application auditing infrastructure is provided through the Web events feature. Web events are a framework for emitting rich informational events during the execution of the application, and channeling that information to one or more configurable providers with rich subscription controls defining how and where the events are sent. ASP.NET provides a base hierarchy of event classes that are used by the framework itself to provide useful information about the execution of an application, and can also be used as a foundation for custom events raised by your application code. One subset of such events that is interesting from a security perspective is the security audit events, which derive from the `System.Web.Management.WebAuditEvent` class, as described in [Figure 9](#).

If you are performing custom application authorization, accessing sensitive resources, or performing any other security-relevant operation, you should consider instrumenting it by creating and raising a custom event derived either from the `WebAuditEvent` class or from the combination of the `WebAuditFailureEvent` and `WebAuditSuccessEvent` classes. [Figure 10](#) shows a custom event that is raised when an .aspx page fails to validate an input parameter. You can capture the desired audit events by configuring a Web events rule to direct all events that derive from a particular audit base class to a specific provider, and configuring the appropriate event throttling and buffering settings based on your requirements. Event throttling allows you to constrain the rate of events generated by your application by dropping events when the rate exceeds your configuration, and buffering allows you to accommodate event spikes by buffering events in memory and flushing them out in an intelligent manner to protect the event delivery channel. It is highly recommended that you use a provider that supports buffering, such as the `System.Web.Management.SqlWebEventProvider`, so that you can use more relaxed throttling settings to avoid losing events while still being able to accommodate a large event flow.

Be aware that normal application operation will generate verbose audit successes that may not be very interesting from a security perspective and may contribute to an overly high event rate that will impact the performance of your application or contribute to a denial of service condition. Because Web events expect that throttling settings will lead to event loss under high event-rate conditions, the sequence number in each event will indicate how many events were actually raised when not all events are captured by the provider.

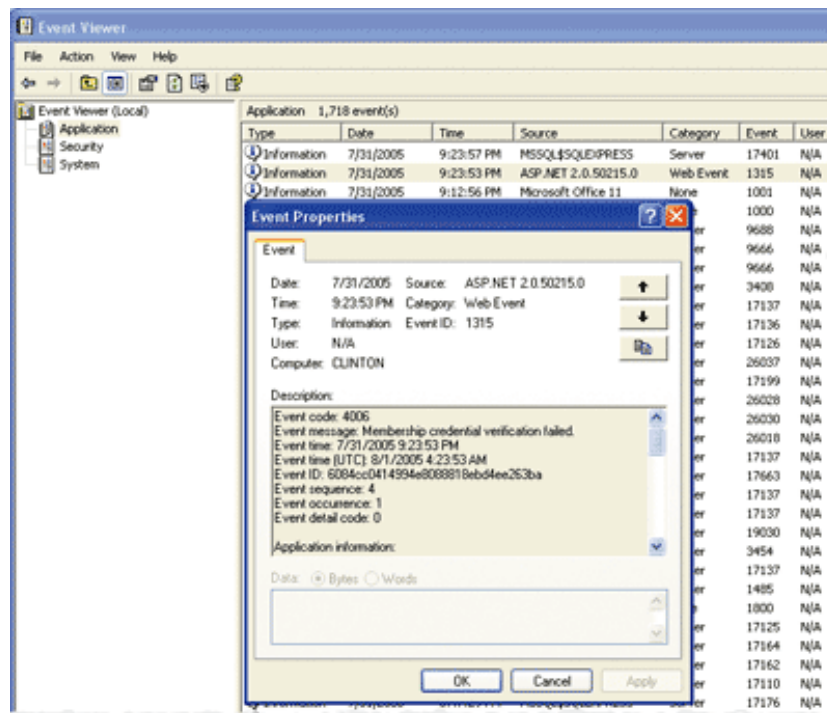


Figure 11 Auditing Event Raised by Invalid Membership Credentials

By default, only audit failure events are captured to the event log provider, and appear at most one per minute to prevent event log flooding. **Figure 11** shows a `WebAuthenticationFailureAuditEvent` raised by the Membership provider when the application attempted to authenticate a user with invalid credentials. To learn more about using Web events, take a look at the Beta2 Quickstart tutorial at [Web Events](#).

[Back to Contents](#) 

Wrapping It Up

IIS 6.0 and ASP.NET provide a solid Web application platform engineered with security in mind. But it's ultimately the responsibility of your application to follow security best practices to avoid introducing potential vulnerabilities. That means you have to pay attention to these best practices during design, development, and deployment to make sure the application has as strong foundation for protecting itself. The practices given in this article should help you evaluate the critical aspects of securing your application, and give you some ideas regarding how to best take advantage of the platform security features provided by IIS and ASP.NET.

Because new classes of attacks and new ways to exploit software are constantly being discovered, following the principles of least privilege, reduced surface area, and defense-in-depth throughout your application's lifecycle can help you mitigate unforeseeable attacks in the future. More secure design practices are available from "[Introduction to Web Application Security](#)" in the patterns & practices guide, a very good source for Web app security guidance that can be used to learn more about specific ideas discussed here. A resource for more ASP.NET 2.0 security best practices can be found at [Security Practices: ASP.NET 2.0 Security Practices at a Glance](#).

[Back to Contents](#) 

Michael Volodarsky is a technical Program Manager on the Web Platform and Tools Team at Microsoft. He owns the core server infrastructure for ASP.NET and IIS and is now focusing on improving the Web application platform in the next-generation Web server, IIS 7.0.



From the [November 2005](#) issue of [MSDN Magazine](#).

[Back to top](#)

QJ: 051103

© 2006 Microsoft Corporation and CMP Media, LLC. All rights reserved; reproduction in part or in whole without permission is prohibited.

[Manage Your Profile](#) | [Legal](#) | [Contact us](#) | [MSDN Flash Newsletter](#)

Microsoft

© 2006 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)