1

Initial Phases of a Web Request

Before the first line of code you write for an `.aspx` page executes, both Internet Information Services (IIS) and ASP.NET have performed a fair amount of logic to establish the execution context for a HyperText Transfer Protocol (HTTP) request. IIS may have negotiated security credentials with your browser. IIS will have determined that ASP.NET should process the request and will perform a hand-off of the request to ASP.NET. At that point, ASP.NET performs various one-time initializations as well as per-request initializations.

This chapter will describe the initial phases of a Web request and will drill into the various security operations that occur during these phases. In this chapter, you will learn about the following steps that IIS carries out for a request:

- The initial request handling and processing performed both by the operating system layer and the ASP.NET Internet Server Application Programming Interface (ISAPI) filter
- How IIS handles static content requests versus dynamic ASP.NET content requests
- How the ASP.NET ISAPI filter transitions the request from the world of IIS into the ASP.NET world

Having an understanding of the more granular portions of request processing also sets the stage for future chapters that expand on some of the more important security processing that occurs during an ASP.NET request as well as the extensibility points available to you for modifying ASP.NET's security behavior.

> *This book describes security behavior primarily for Windows Server 2003 running IIS6 and ASP.NET. Due to differences in capabilities between IIS5/5.1 and IIS6, some of what is described is not available or applicable when running on Windows 2000/XP. Differences in behavior between versions of IIS are noted in some cases.*

IIS Request Handling

The initial processing of an HTTP request on Windows Server 2003 occurs within both IIS and a supporting protocol driver. As a result, depending on the configuration for IIS, a request may never make it far enough to be processed by ASP.NET. The diagram in Figure 1-1 shows the salient portions of IIS and Windows Server 2003 that participate in request processing.

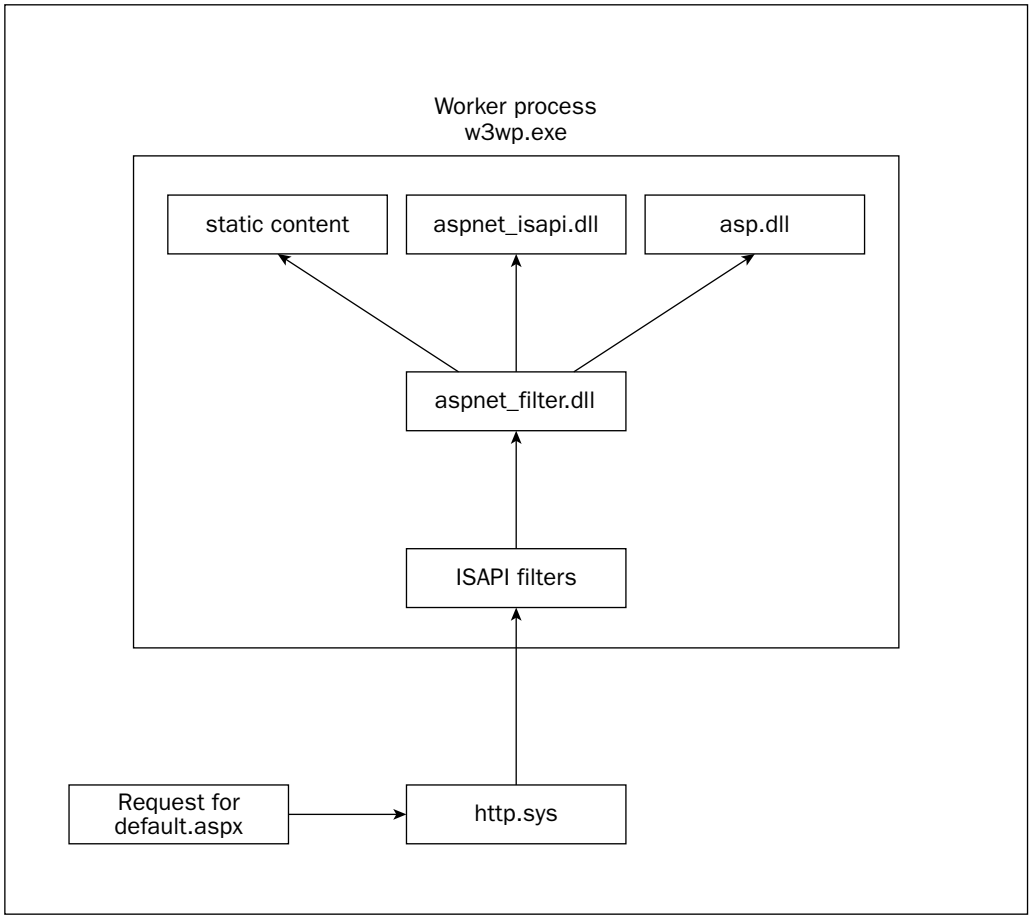

Figure 1-1

A request must first make it past the restrictions enforced by the kernel mode HTTP driver: http.sys. The request is handed off to a worker process where it then flows through a combination of the internal request processing provided by IIS and several ISAPI filters and extensions. Ultimately, the request is routed to the appropriate content handler, which for ASP.NET pages is the ASP.NET runtime's ISAPI extension.

Http.sys

When an HTTP request is first received by Windows Server 2003, the initial handling is actually performed by the kernel-mode HTTP driver: `http.sys`. The kernel mode driver has several Registry switches that control the amount of information allowed in a request URL. By default the combined size of the request URL and associated headers — any query string information on the URL, and individual headers sent along with the request, such as cookie headers — must not exceed 16KB.

Furthermore, no individual header may exceed 16KB. So, for example, a user agent could not attempt to send a cookie that is larger than 16KB (although for other reasons, a 16KB cookie would be rejected by ASP.NET anyway). Under normal circumstances the restrictions on headers and on the total combined size of the request URL and headers is not a problem for ASP.NET applications. However, if your application depends on placing large amounts of information in the URL — perhaps for HTTP-based .asmx Web Services — then the length limit enforced by `http.sys` may come into play.

Any application that depends on excessively long request URLs or request headers should, if at all possible, have its logic changed to transmit the information through other mechanisms. For a Web Service, this means using Simple Object Access Protocol (SOAP) headers to encapsulate additional request data. For a website, information needs to be sent using a `POST` verb, rather than a `GET` verb.

The kernel mode driver restricts the number of path segments in a URL and the maximum length for any individual path segment. Examine the following URL:

```
http://yoursite/application1/subdirectory2/resource.aspx
```

The values `application1`, `subdirectory2`, and `resource.aspx` represent individual path segments. By default, http.sys disallows URLs that have more than 255 path segments and URLs where the length of any single path segment exceeds 260 characters. These constraints are actually pretty generous, because in practice developers normally do not need large number of path segments, even for applications with a fair amount of directory nesting. The requested page in the previous example, `resource.aspx`, is considered a path segment and is subject to the same length restrictions as any portion of the URL. However, if there were query string variables after `resource.aspx`, the length of the query string variables would apply only against the overall 16KB size restriction on the combined size of URL plus headers. As a result, you can have query string variables with values that are greater than 260 characters in length.

One reason for these size limits is that a number of hack attacks against web servers involve encoding the URL with different character representations. For example, an attacker may attempt to bypass directory traversal restrictions by encoding periods like this:

```
http://yoursite/somevirtualdirectory/%2E%2E/%2E%2E/%2E%2E/boot.ini
```

As you can see, encoding characters bloats the size of the URL, so it is reasonable to assume that excessively long URLs are likely due to hacker attempts.

To give you a concrete example of `http.sys` blocking a URL, consider a request of the following form:

```
http://localhost/12345678901234567890123456789 0etc.../foo.htm
```

The sequence `1234567890` is repeated 26 times in the URL. Because the path segment is exactly 260 characters though, `http.sys` does not reject the request. Instead, this URL results in a 404 from IIS because there is no `foo.htm` file on the system.

However, if you add one more character to this sequence, thus making the path segment 261 characters long, an HTTP 400 - Bad Request error message is returned. In this case, the request never makes it far enough for IIS to attempt to find a file called `foo.htm`. Instead, `http.sys` rejects the URL and additional IIS processing never occurs. This type of URL restriction reduces the load on IIS6, because IIS6 does not have to waste processor cycles attempting to parse and process a bogus URL.

This raises the question of how a web server administrator can track URL requests are being rejected. The `http.sys` driver will log all errors (not just security-related errors) to a special HTTP error log file. On Windows Server 2003, inside of the `%windir%\system32\LogFiles` directory, there is an `HTTPERR` subdirectory. Inside of the directory one or more log files contain errors that were trapped by `http.sys`. In the case of the rejected URLs, a log entry looks like:

```
2005-03-13 22:09:50 127.0.0.1 1302 127.0.0.1 80 HTTP/1.1 GET /1234567890....htm 400
- URL
```

For brevity the remainder of the `GET` URL has been snipped in the previous example; however, the log file will contain the first 4096 bytes of the requested URL. In this example, the value `URL` at the end of the log entry indicates that parsing of the URL failed because one of the path segment restrictions was exceeded.

If the URL is larger than 16KB, the log entry ends with `URL_Length`, indicating that the allowable URL length had been exceeded. An example of such a log entry is:

```
2005-03-13 23:02:53 127.0.0.1 1086 127.0.0.1 80 HTTP/0.0 GET - 414 -
URL_Length
```

For brevity, the URL that caused this is not included because a 16KB long URL would not be particularly interesting to slog through. Remember that form posts and file uploads also include a message body that usually contains the vast majority of the content being sent to the web server. Because `http.sys` only checks the URL and associated headers, it does not perform any validation on the size of the message body. Instead it is ASP.NET that is responsible for limiting the size of raw form post data or file uploads.

A subtle point about the previous discussion is that some of the restrictions `http.sys` enforces are based on number of characters, while other restrictions are based on byte size. In the case of path segments, the restrictions are based on number of characters, regardless of the underlying character set. However, for the 16KB size restrictions, the actual URL or header allowed depends heavily on the characters in the URL or headers. If a URL or header contains only standard ASCII characters, a 16KB size limit equates to 16384 characters. However, if a URL or header contains characters other than standard ASCII characters, converting from byte size to character length becomes a bit murkier.

Because `http.sys` processes URLs as UTF-8 by default, and UTF-8 characters consume between 1 and 3 bytes in memory, an allowable URL length could be anywhere from roughly 5461 characters to 16384 characters. A general rule of thumb when using non-ASCII characters though is to assume 2 bytes per character if there is extensive use of Unicode characters, which equates to a maximum URL length (including query string variables) of 8192 characters.

The character length and byte size restrictions enforced by `http.sys` can be modified by adding `DWORD` values underneath the following Registry key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\HTTP\Parameters
```

The specific Registry settings that govern the behavior just discussed are listed in the following table. Also, a server reboot is required after you change any of the following settings.

Registry Setting Value Name	Description
`MaxFieldLength`	By default, an individual header can be up to 16KB in size. Change this setting to limit the size of any individual HTTP header. A request URL, including query string information, is also restricted in size by this setting. The allowed range of values is 64–65534 bytes.
`MaxRequestBytes`	By default, the combined size of the request URL, including query string, plus its associated HTTP headers cannot exceed 16KB. The allowed range of values is 256–16777216 bytes.
`UrlSegmentMaxCount`	By default, no more than 255 path segments are allowed in a URL. The allowed range of values is 0–16383 segments.
`UrlSegmentMaxLength`	By default, an individual path segment cannot be longer than 260 characters. The slashes that delimit each path segment are not included when computing a path segment's character length. The allowed range of values is 0–32766 characters.

In earlier versions of IIS, the URLScan security tool (available by searching `microsoft.com/technet`*) provides similar protections for restricting URLs. Most of the security functionality of URLScan was incorporated into* `http.sys` *and IIS6. There are a few small features that are only available with URLScan though, the most interesting one being URLScan's ability to remove the server identification header that IIS sends back in HTTP responses.*

aspnet_filter.dll

After `http.sys` is satisfied that the request is potentially valid, it passes the request to the appropriate worker process. In IIS6 multiple application pools can be running simultaneously, with each application essentially acting as a self-contained world running inside of an executable (`w3wp.exe`). Within each worker process, IIS carries out a number of processing steps based on the ISAPI extensibility mechanism. Even though ASP.NET is a managed code execution environment, it still depends on the ISAPI mechanism for some initial processing.

When ASP.NET is installed on a web server, it registers an ISAPI filter with IIS. This filter (`aspnet_filter.dll`) is responsible for two primary tasks:

- Managing cookieless tickets by converting them into HTTP headers
- Preventing access over the Web to protected ASP.NET directories

You can see the set of all ISAPI filters that are registered in IIS by using the IIS MMC, right-clicking the Web Sites node, and then clicking on the ISAPI Filters tab in the dialog box that opens. In Figure 1-2, you can see that there is currently only one ISAPI filter registered by default — the ASP.NET filter. Depending on your machine, you may see additional filters that provide services such as compression or that support Front Page extensions.

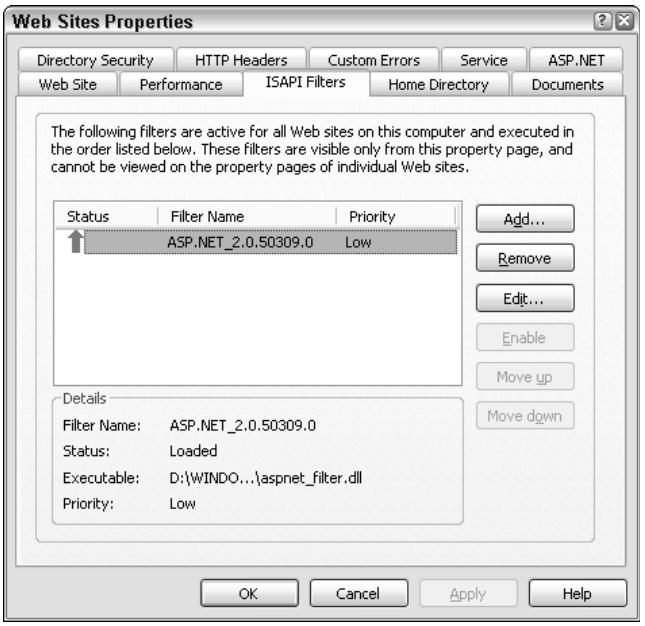

Figure 1-2

By default ASP.NET registers the filter with a Low priority, which means that other filters with higher priorities will have the opportunity to inspect and potentially modify each incoming request. This makes sense because if, for example, you are running a filter that decompresses incoming HTTP content, you would want this type of operation to occur prior to ASP.NET carrying out security logic based on the request's contents.

The ASP.NET filter handles two ISAPI filter notifications: `SF_NOTIFY_PREPROC_HEADERS` and `SF_NOTIFY_URL_MAP`. This means the filter has the opportunity to manipulate the request prior to IIS attempting to do anything with the HTTP headers, and the filter has the opportunity to perform some extra processing while IIS is converting the incoming HTTP request into a request for a resource located at a specific physical path on disk.

Processing Headers

The ASP.NET filter inspects the request URL, looking for any cookieless tickets. In ASP.NET 2.0, cookieless tickets are supported for session state (this was also available in 1.1), forms authentication (previously available as part of the mobile support in ASP.NET) and anonymous identification (new in ASP.NET 2.0). A sample URL with a cookieless session state ticket is shown here:

```
http://localhost/inproc/(S(tuucni55xfzj2xqx1mnqdg55))/Default.aspx
```

ASP.NET reserves the path segment immediately after the application's virtual root as the location on the URL where cookieless tickets are stored. In this example, the application was called `inproc`, so the next path segment is where ASP.NET stored the cookieless tickets. All cookieless tickets are stored within an outer pair of parentheses. Within these, there can be a number of cookieless tickets, each starting with a single letter indicating the feature that consumes the ticket, followed by a pair of parentheses that contain the cookieless ticket. Currently, the following three identifiers are used:

- **S** — Cookieless ticket for session state
- **A** — Cookieless ticket for anonymous identification
- **F** — Cookieless ticket for forms authentication

However, the ASP.NET filter does not actually understand any of these three indentifiers. Instead, the filter searches for the character sequences described earlier. Each time it finds such a character sequence, it removes the cookieless ticket, the feature identifier and the containing parentheses from the URL and internally builds up a string that represents the set of cookieless tickets that it found. The end result is that all cookieless tickets are removed from the URL before IIS attempts to convert the URL into a physical path on disk. Therefore, IIS doesn't return a 404 error even though there clearly is no directory on disk that starts with `(S)`.

After the filter removes the tickets from the URL, it still needs some way to pass the information on to the ASP.NET runtime. This is accomplished by setting a custom HTTP header called `ASPFILTERSESSIONID`. The name is somewhat misleading because it is a holdover from ASP.NET 1.1 when the only cookieless ticket that was supported (excluding mobile controls and the cookieless forms authentication support that was part of the mobile controls) was for session state. With ASP.NET 2.0, though, there are obviously a few more cookieless features integrated into the product. Because the underlying logic already existed in the ISAPI filter, the old header name was simply retained.

You can actually see the effect of this header manipulation if you dump the raw server variables associated with an ASP.NET request. As an example, for an application that uses both cookieless session state and cookieless forms authentication, the URL after login may look as follows:

```
http://localhost/inproc/(S(sfeisy55occclkmlkcwtjz55)F(jbZ....guo1))/Default.aspx
```

For brevity the majority of the forms authentication ticket has been removed. However, the example shows cookieless tickets for session state and forms authentication in the URL. If you were to dump out the server variables on a page, you would see the following header:

```
HTTP_ASPFILTERSESSIONID=S(sfeisy55occclkmlkcwtjz55)F(jbZ....guo1)
```

Hopefully, this sample makes it clearer how the unmanaged ISAPI ASP.NET filter transfers cookieless tickets over to the ASP.NET runtime. Within the ASP.NET runtime, the HTTP modules that depend on these tickets have special logic that explicitly looks for this HTTP header and parses out the ticket information for further processing (for example, setting up the session, validating forms authentication credentials, and so on).

Blocking Restricted Directories

After the filter processes any cookieless tickets, the filter has IIS normalize the request URL's representation. This is necessary because the filter enforces the restriction that browser users cannot request any type of content from the protected directories in ASP.NET 2.0. Because ASP.NET 2.0 introduced new "content" that in reality consists of code, data, resources, and other pieces of information, it is necessary to prevent access to this information via a browser. The filter prevents access by scanning the normalized URL, looking for one of the following paths:

- **/bin** — Compiled assemblies referenced by the application
- **/app_code** — Source code files with classes referenced elsewhere in an application
- **/app_data** — Data files such as `.xml`, `.mdb`, or `.mdf` files
- **/app_globalresources** — Resources that are globally accessible throughout an application
- **/app_localresources** — Resources that are applicable to a specific directory
- **/app_webreferences** — WSDL files and compiled artifacts for Web Services
- **/app_browsers** — Browser capability files for determining browser functionality

If the filter finds a path segment with one of these paths, the filter returns an error to IIS, which is converted into a 404 response and returned to the browser. For example, if a web server has a directory immediately under `wwwroot` called `app_data` with an HTML file called `foo.htm`, requesting the following URL still result in a 404 even though the file does exist on the file system.

```
http://localhost/app_data/foo.htm
```

There had been some discussion at one point around having the filter perform a broad blocking of any URLs that contained the characters `/app_` at the beginning of a path segment. However, this decision was avoided because some developers may have already been using such a naming prefix in their directory structures. If at all possible, it is recommended that developers move away from naming any directories with the `/app_` prefix. In a future release of ASP.NET, the filter may support blocking any paths that start with these characters — not just the specific set of reserved directories in ASP.NET 2.0.

If you have valid reasons for creating directory structures on disk with any of the reserved names noted earlier, you can disable the filter's directory blocking behavior (although for security reasons this is clearly not recommended). Registry settings to control the directory blocking behavior can be added as DWORD values underneath the following Registry key:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\ASP.NET
```

After changing any of the settings shown in the following table, run `iisreset` to recycle the worker processes. This forces `aspnet_filter.dll` to read the new Registry settings when the filter is initialized in a new worker process.

Registry Setting Value Name	Description
`StopBinFiltering`	Set this value to 1 to stop the filter from blocking requests to paths that include `/bin`. This setting will affect all ASP.NET 1.1 and 2.0 applications on the server.
`StopProtectedDirectoryFiltering`	Set this value to 1 to stop the filter from blocking requests to reserved ASP.NET directories that include a path starting with `/app_`. Because this setting is new to ASP.NET 2.0, it will only affect all ASP.NET 2.0 applications on the server.

Setting either one of these Registry settings will affect all of your websites. There is no mechanism to selectively turn off directory blocking for only specific applications or specific websites.

Dynamic versus Static Content

After a request has flowed through all of the ISAPI filters configured for a website, IIS decides whether the requested resource is considered static content or dynamic content. This decision really depends on whether a custom ISAPI extension has been configured and associated with the file extension of the requested resource. For example, if you were to request `http://localhost/foo.htm`, in the default configuration of IIS, the `.htm` extension is registered as a type of static content server directly by IIS.

The configuration of static versus dynamic content is determined by a combination of settings in IIS6:

- MIME type mappings
- File extension to ISAPI extension mappings
- The presence of wildcard application mappings (if any)

MIME Type Mappings

IIS6 is configured with several well known static file extensions in its list of Multipurpose Internet Mail Extensions (MIME) type mappings. The reason that MIME type mappings are so important in IIS6 is that without a MIME type mapping, an HTTP request for a file results in a 404 error, even if the file does exist on the file system. For example, if a text file, `foo.xyz`, exists at the root of a website, requesting `http://localhost/foo.xyz` results in a 404.

However, the web server's allowable MIME types can be edited to allow IIS6 to recognize `.xyz` as a valid file extension. In Figure 1-3, the IIS6 MMC is shown being used to register `.xyz` as a valid file extension.

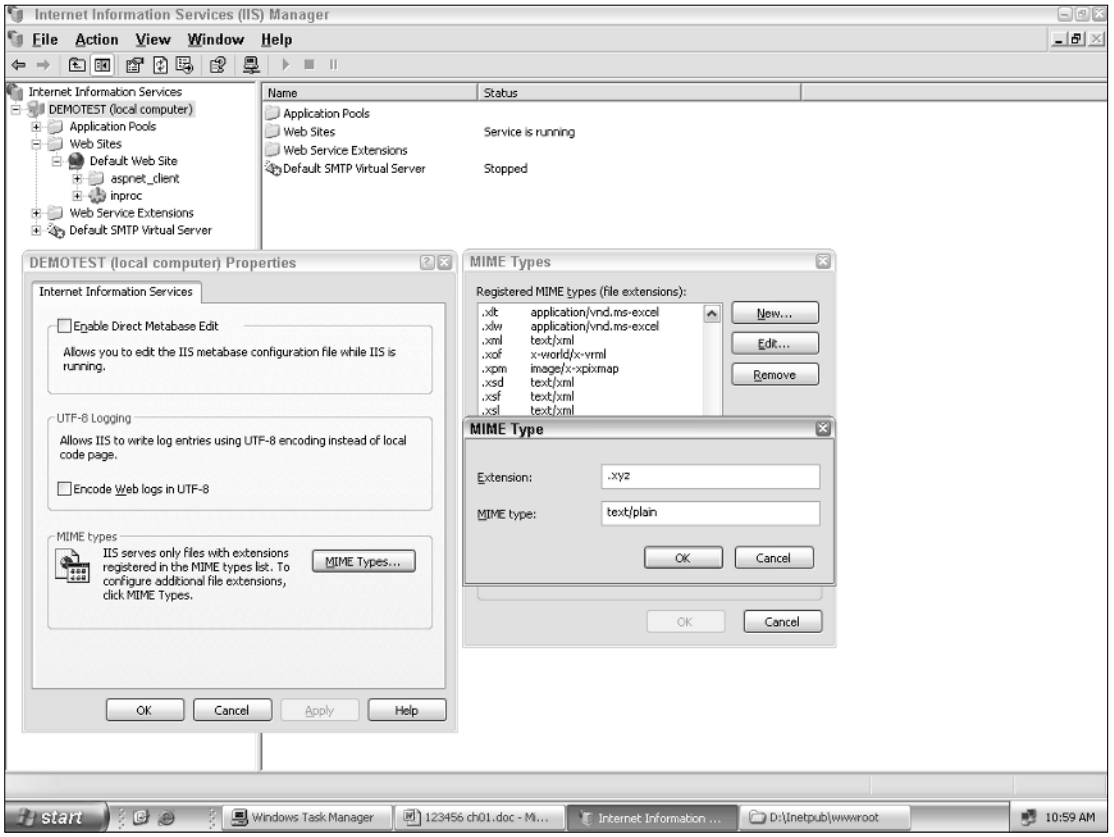

Figure 1- 3

Right clicking the computer node and selecting Properties pulls up a dialog box that allows you to configure MIME types. Click the MIME Types button to access the Mime Types dialog box, where you can click the New button to add a new MIME type. For this example, the `.xyz` file extension was added as a being a text type.

You need to `iisreset` for the changes to take affect. When the web server is running again, a request for `http://localhost/foo.xyz works`, and IIS6 returns the file's contents.

ISAPI Extension Mappings

Because a web server that serves only static files would be pretty useless in today's web, ISAPI extension mappings are available for serving dynamically generated content. However, ISAPI extensions can also be used to carry out server-side processing on static file content. For example, there are ISAPI extensions for processing server-side include files. In practice though, ISAPI extensions are typically used for associating file extensions with Dynamic Link Libraries (DLLs) that carry out the necessary logic for executing code and script to dynamically generate page output.

You can see the list of ISAPI extensions that are mapped to a website with the following steps:

1. Right-click the application's icon in the IIS6 MMC.
2. Select properties.
3. In the Directory tab of the dialog box that pops up, click the Configuration button.
4. In the Mappings tab of the dialog box that pops up, a list box shows all application extensions currently mapped for the web application.

In Figure 1-4, the current application has mapped the `.aspx` file extension to a rather lengthy path that lives somewhere in the framework installation directory.

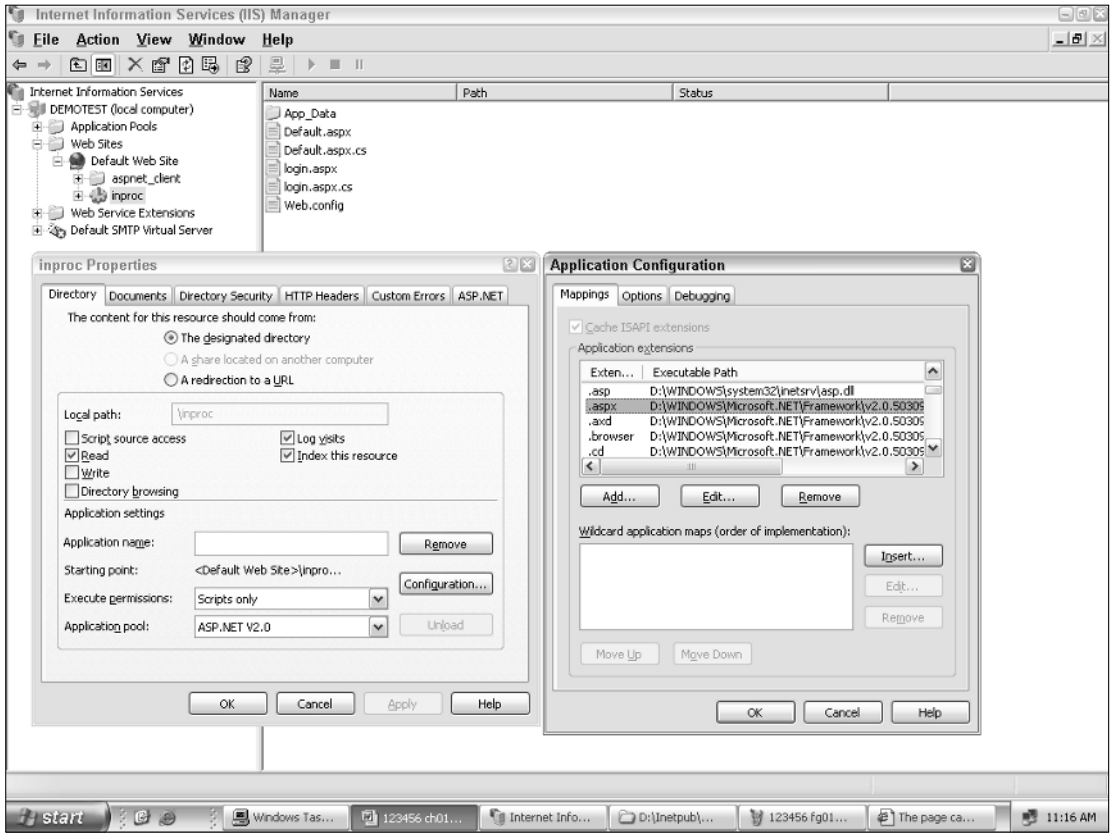

Figure 1-4

The path is too long to see without scrolling around, but it points at the following directory location:

```
%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_isapi.dll
```

Depending on where you installed the operating system on your machine, the location of `%windir%` will vary.

When IIS receives a request for a file, if the file extension for that request is mapped to an ISAPI extension, IIS routes the request to the mapped ISAPI extension instead of consulting the list of MIME types and serving the file as static content. In the case of the `.aspx` file extension, the request is routed to `aspnet_isapi.dll`, which contains the code that bootstraps the ASP.NET runtime and allows ASP.NET pages to run.

If you scroll around a bit through the various application extensions, you can see that there are a large number of mapped extensions. Clicking the Executable Path column sorts the extensions and makes it easier to see which file extensions are currently mapped to the ASP.NET ISAPI extension. Most of the extensions that start with the letter a should be familiar to varying degrees (everyone who writes HTTP handlers raise your hand!). Several other file extensions are probably familiar to you from working with tools like Visual Studio or SQL Server, but it may not make sense why these file extensions are now mapped to the ASP.NET ISAPI extension.

For example, the various Visual Studio project extensions (`.csproj`, `.vbproj`) are mapped to `aspnet_isapi.dll`. Simiarly, SQL Server database extensions (`.ldf` and `.mdf`) are mapped to `aspnet_isapi.dll`. From experience though, you know that your ASP.NET web servers have not been processing project files or opening database files and pretending to be a database engine.

This leads to another approach of using ISAPI extensions. Not only do ISAPI extensions parse and process files that are mapped to them, but ISAPI extensions can also be configured to handle other file types for specific purposes. When ASP.NET is installed, file extensions for files that commonly occur within a developer's ASP.NET project are mapped to the ASP.NET ISAPI extension. Because XCOPY deployment is an easy way to move an ASP.NET application from a developer's desktop onto a web server, there can be a number of files within the structure of an ASP.NET project that the developer does not want served to the Internet at large. By mapping these file extensions to `aspnet_isapi.dll`, IIS will pass requests for these file types to the ASP.NET runtime. Because ASP.NET has a parallel configuration system that maps file extensions to specific processing logic (`.aspx` pages are executed by the ASP.NET page handler), ASP.NET can choose to do something other than executing the requested file. In the case of file extensions like `.csproj` or `.mdf`, ASP.NET has a special handler that will deny access to files of this type and return an error to that effect. This technique will be revisited later in the chapter when the default handler mappings for ASP.NET are discussed.

Throughout this discussion there has been the implicit assumption that after a mapping between a file extension and an ISAPI extension is established, dynamic content will start working. Although this was the case for IIS5 and IIS5.1, IIS6 introduced an extra layer of protection around ISAPI extensions. On IIS6, an administrator must take some kind of explicit action to allow an ISAPI extension to operate. If IIS6 is installed on a Windows Server 2003 machine in its most basic configuration, even though ASP.NET bits exist on the machine, requests to `.aspx` pages will always fail with a 404 error.

The reason for this is that IIS6 has the ability to enable and disable individual ISAPI extension DLLs. If you use the Manage Your Server Wizard in Windows Server 2003, it will automatically reenable the ASP.NET1.1 ISAPI extension for you when you configure the server in the Application Server role. As a result, when the 2.0 version of the framework is installed on top of it, the ASP.NET 2.0 ISAPI extension will be enabled as well.

However, if you install the 2.0 version of the framework but are still receiving 404 errors, you need to enable the ASP.NET ISAPI extension. Figure 1-5 shows the Web Service Extensions configuration window in the IIS MMC. Right-click the ASP.NET extension to access the option to enable the extension.

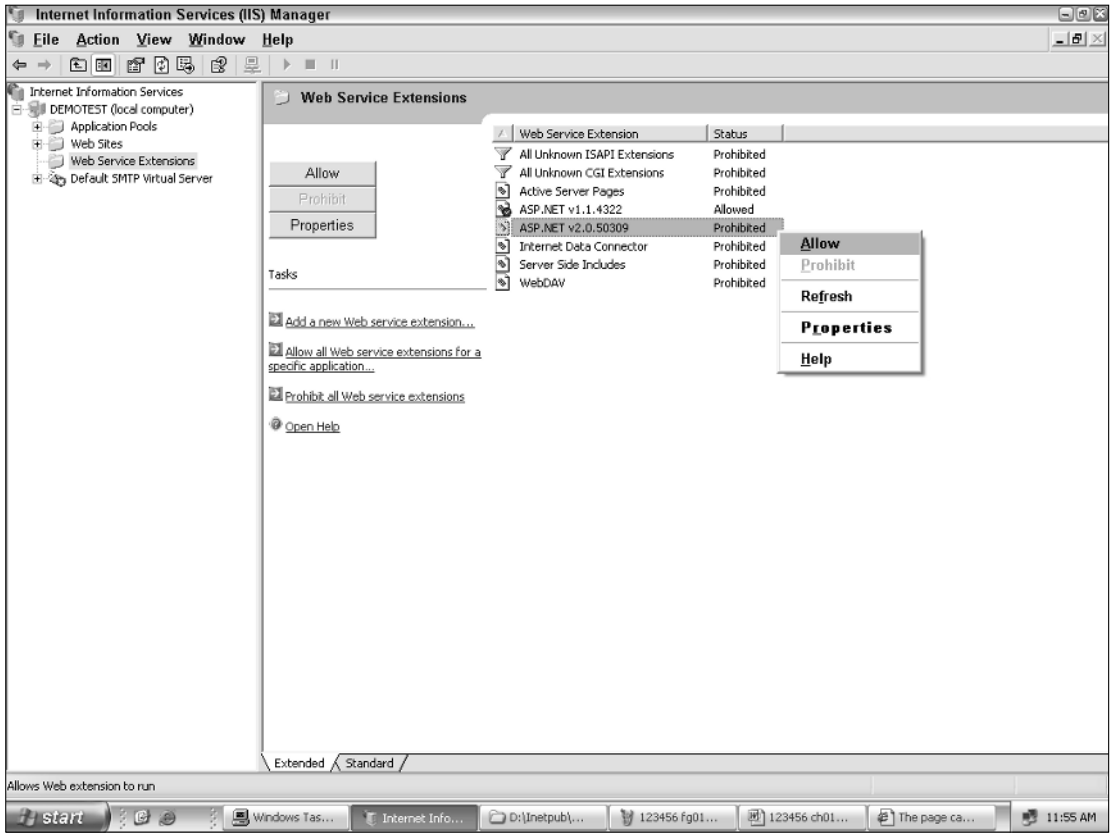

Figure 1-5

Aside from causing premature gray hair for developers and administrators wondering why a perfectly good ASP.NET application is dead in the water, the ISAPI extension lockdown capability does serve two useful purposes:

- If the web server is not intended to ever serve dynamic ASP.NET content, disabling ISAPI extensions is an easy and effective way to lock down the server.
- With the release of ASP.NET 2.0, you can use this feature to disable the ASP.NET 1.1 ISAPI extension. For example, if you want to ensure that only ASP.NET 2.0 applications are deployed onto a specific web server, you can disable the ASP.NET 1.1 extension on that server.

Wildcard Application Mappings

IIS6 introduced the concept of wildcard application mappings. With IIS5/5.1, customers were asking for the ability to map all requests for content to a specific ISAPI extension. However, the only way to accomplish this prior to IIS6 was to laboriously map each and every file extension to the desired ISAPI extension. Also, after the request was routed to the ISAPI extension, the ISAPI extension was responsible for completing the request. There was no mechanism for passing the request to other ISAPI extensions or back to IIS.

With IIS6, it is now possible to set up rules (aka wildcard application maps) that route all HTTP requests to one or more ISAPI extensions. The set of wildcard application mappings can be prioritized, so it is possible to have a chain of wildcard mappings. IIS6 also includes a new API for ISAPI extensions to route a request out of an extension and back to IIS6. The net result is that with IIS6 and ASP.NET 2.0, it is possible to have a request for a static file flow through the first portion of the ASP.NET pipeline, and then have the request returned to IIS6, which subsequently serves the file from the file system.

Out of the box though, ASP.NET 2.0 does not configure or use any wildcard application mappings. ASP.NET 2.0 does include though the necessary internal changes required to flow a request back out to IIS6. As a result, ASP.NET 2.0 has this latent ability to integrate with and use wildcard application mappings for some very interesting scenarios. As mentioned earlier, it is possible for an ISAPI extension to perform some processing for a requested file without actually understanding the requested file format. An interesting new avenue for integrating ASP.NET 2.0 with static files and legacy ASP code is discussed later in this book in Chapter 6, "Integrating ASP.NET Security with Classic ASP," The techniques in that chapter depend on the wildcard application mapping functionality of IIS6.

aspnet_isapi.dll

After a request reaches `aspnet_isapi.dll` ASP.NET takes over responsibility for the request. IIS6 itself knows nothing about managed code or the .NET Framework. On the other hand, the core processing classes in ASP.NET (`HttpApplication` and the specific handlers that run `.aspx` pages, `.asmx` Web Services, and so on) do not possess the ability to reach out and directly consume an HTTP request. Although the vast majority of ASP.NET is managed code, the ISAPI extension plays a critical role in bridging the native and managed code worlds.

The responsibilities of the ISAPI extension fall into two broad areas:

- Starting up an application domain so that managed code associated with an application can run
- Setting up the security context for each request and then passing control over to the managed portion of ASP.NET

Understanding some of the important portions of application domain startup is important for later discussions on trust levels and configuration. Information about the per-request initializations and handoff will be covered in Chapter 2.

ASP.NET includes several classes in the `System.Web.Hosting` namespace that can be used by applications that want to host ASP.NET. If you use the file-based web project option in Visual Studio 2005, you are using a standalone executable (`WebDev.WebServer.exe` located in the framework install directory) to host ASP.NET. Also, if you search on the Internet several articles and sources demonstrate how to write console and Winforms applications to host ASP.NET. However, most ASP.NET developers are writing web applications and expect their applications to be hosted on a web server. As a result, you can think of `aspnet_isapi.dll` and its supporting managed classes as the default implementation of an ASP.NET host.

Starting Up an Application Domain

All managed code in the .NET Framework needs to run within an application domain. Before ASP.NET can start the HTTP pipeline and run a page, the ISAPI extension must ensure that an application domain has been instantiated and initialized. In ASP.NET, each application, as configured in the IIS MMC, maps to a separate application domain in the managed world. Figure 1-6 shows a web server with a default website, and one IIS application configured beneath the root of the default website.

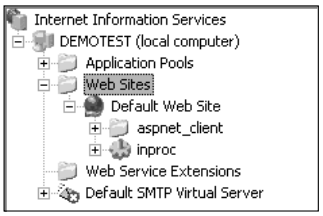

Figure 1-6

The ASP.NET ISAPI extension will ensure that an application domain is created for ASP.NET during the first request for a page in the default website. If another request were received for a page within the web application called `inproc`, `aspnet_isapi.dll` would create a second application domain because `inproc` is configured as a separate application. Overall, this means that within a single IIS6 worker process, any number of configured IIS applications, and thus independent application domains, can be running. It is the responsibility of the ISAPI extension to route each incoming HTTP request to the appropriate application domain. Isolating the different applications into separate application domains gives ASP.NET the flexibility to perform some of the following tasks:

- Maintain separate security configurations for each application domain
- Enforce different trust level restrictions in each application domain
- Monitor and if necessary recycle application domains without affecting other application domains

Starting up an application domain involves several processing steps. After a new application domain has been created, the ISAPI extension carries out the following steps, listed in order of their occurrence:

1. Establish the identity for application domain initialization.
2. Verify directory access/existence and initializing directory information.
3. Set the trust level for the application domain.
4. Set the locations of assemblies.
5. Obtain the auto-generated machine key.
6. Initialize the ASP.NET compilation system.

Establishing Identity

Prior to the ISAPI extension performing any other initialization work, it ensures that the correct security identity is established. The identity used for initialization is one of the following:

1. If the application is running from a local disk, and there is no `<identity />` tag with an application impersonation identity, then the identity of the worker process is used. Under IIS6 this would be `NT AUTHORITY\NETWORK SERVICE`. On older versions of IIS, the identity would the local ASPNET machine account. Even if the current thread is running with other security credentials established by IIS, the ISAPI extension will temporarily revert to using the process identity.
2. If the application has an `<identity />` tag that enables impersonation, and there is an explicit username and password configured (usually referred to as application impersonation), then initialization will run as the application impersonation identity. ASP.NET will attempt to create a security token for this identity, calling `LogonUser` in sequence for each of the following logon types until a logon succeeds: `BATCH`, `SERVICE`, `INTERACTIVE`, `NETWORK_CLEARTEXT`, and `NETWORK`.
3. If the application was configured to run off of a UNC share, and there is no application impersonation identity, initialization will run with the configured UNC credentials.

Initializing Directory Information

An ASP.NET application depends on a number of directories for the application to execute properly. The extension will first ensure that the physical application directory exists. If the application directory does not actually exist, or if the current security identity does not have read access to the application directory, the extension returns an error stating that the server could not access the application directory.

Next, ASP.NET initializes the application-relative data directory information. In the v2.0 of the Framework, ADO.NET supports the ability for applications to set application-relative path information to a data file. This allows applications, such as ASP.NET applications, to deploy SQL Server files in an application-relative location (the `App_Data` directory). The application can then reference the database using a standard connection string syntax that does not change even when the underlying file structure is moved. For all of this magic to work though, ASP.NET must set an application domain variable, `DataDirectory`, with the proper physical path information so that ADO.NET can correctly resolve relative directories in connection strings. As part of application domain startup, ASP.NET determines the full physical path to the data directory and stores it in the `DataDirectory` application domain variable.

Any code can query an application domain and retrieve this application domain variable just by calling `AppDomain.CurrentDomain.GetData("DataDirectory")`. Because storing physical paths could lead to an information disclosure, ASP.NET also tells the framework to demand `FileIOPermissionAccess.PathDiscovery` from any callers. In practice, this means any ASP.NET application running at Low trust or higher can inspect this variable (trust levels and how they work are covered in Chapter3, "A Matter of Trust.")

The last major piece of directory related initialization involves the code generation directories used by ASP.NET. Most ASP.NET applications cannot generate page output based solely on `.aspx` pages that are deployed to a web server. ASP.NET usually has to take additional steps to auto-generate classes (page classes, user control classes, and so on) that are derived from the classes a developer works with in code-behind files. In ASP.NET 2.0 there is a wide array of other auto-generated and auto-compiled artifacts

beyond just page classes. For example, ASP.NET 2.0 dynamically generates a class definition based on the `<profile />` configuration element and then compiles the resulting class definition. For all these types of activities, ASP.NET needs a default location for generated code as well as the compiled results of the auto-generated code.

By default, during application domain initialization, ASP.NET will attempt to create an application specific code-generation (or codegen for short) directory structure at the following location:

```
%windir%\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files\appname
```

As noted earlier, your Windows path will vary, and the final shipping version of the framework will have a different version number. The final portion of this directory path will reflect the name of the ASP.NET application.

By default, when the framework is installed, the local machine group IIS_WPG, the local machine account ASPNET, and the NT AUTHORITY\NETWORK SERVICE accounts are granted read and write access (in addition to other security rights) to this temporary directory. As a result, the current security identity normally has rights to create an application specific code-generation directory. If the current security identity does not have read and write access to the Temporary ASP.NET Files directory, then ASP.NET will return an exception to that effect.

If you are running ASP.NET as an interactive user, ASP.NET will fall back and use the operating system's temporary directory as the root beneath which it will create code-generation directories. On Windows Server 2003, the temporary directory structure is rooted at `%windir%\TEMP`. You will likely encounter this situation if a developer uses a file-based web while developing in Visual Studio 2005. File-based webs use the standalone Cassini web server for running ASP.NET applications and Cassini runs as the current interactive user. If the interactive user does not have read and write access to the Temporary ASP.NET Files directory (for example the interactive user is not a machine administrator or a member of Power Users), then the operating system's temporary directory structure would be used instead. Again though, this fallback behavior is limited to only the case where the ASP.NET host is running as an interactive user. On most production web servers, this will never be the case.

Setting the Trust Level

As a quick recap of code access security (CAS) concepts, remember that the .NET Framework can use four levels of code access security policies:

1. Enterprise
2. Machine
3. User
4. Application domain

The first three levels of CAS policy can be configured and maintained by administrators to ensure a consistent set of CAS restrictions. However, an administrator normally has no ability to configure or enforce application domain CAS restrictions.

ASP.NET 1.1 introduced the concept of trust levels and exposed a configuration element (`<trust />`) as well as Extensible Markup Language (XML) text files that contain the actual definitions of various ASP.NET trust levels. Later in the book in Chapter 3 the specifics of the ASP.NET trust level settings will

be discussed in more detail. However, trust levels are introduced at this point of the discussion because application domain initialization is where ASP.NET loads and applies the appropriate trust level information. After you understand how ASP.NET trust levels work, the knowledge that an ASP.NET trust level is converted into and applied as an application domain policy very early in the lifetime of an application domain helps to explain some of the more obscure security errors customers may encounter.

In practice, many folks are probably unaware of ASP.NET's ability to apply an application domain policy, and instead their websites run in full trust. Partly this is due to the fact that both ASP.NET 1.1 and ASP.NET 2.0 set the ASP.NET trust level to full by default. Full trust means that the .NET Framework allows user-authored code the freedom to call any API without any security restrictions.

After ensuring that the required directories are available, ASP.NET checks the trust level setting in configuration that is found in the `<trust />` configuration section. Based on the configured trust level, ASP.NET loads the appropriate trust policy configuration file from the following directory:

```
%windir%\Microsoft.NET\Framework\v2.0.50727\CONFIG
```

The contents of the trust policy file are modified in memory to replace some of the string replacement tokens that are present in the physical policy files. The end result of this processing is a reference to a `System.Security.Policy.PolicyLevel` instance that represents the desired application domain security policy. ASP.NET then applies the policy level to the application domain by calling `System.AppDomain.CurrentDomain.SetAppDomainPolicy`.

This processing is one of the most critical steps taken during application domain initialization because prior to setting the application domain's security policy, any actions taken by ASP.NET are running in full trust. Because a full trust execution environment effectively allows managed code to call any API (both managed APIs and native APIs), ASP.NET intentionally limits the initialization work it performs prior to setting the application domain's security policy. Looking back over the initialization work that is completed prior to this step, you can see that ASP.NET has not actually called any user-supplied code up to this point. All of the initializations are internal-only checks and involve only framework code.

With the application domain's permission policy established though, any subsequent initialization work (and of course all per-request processing) that calls into user-supplied code will be restricted by the application domain policy that ASP.NET has applied based on the contents of a specific ASP.NET trust policy configuration file.

An important side effect from establishing the trust level is that any calls into the configuration system from this point onwards are subject to the security restrictions defined by the trust level. Configuration section handlers are defined in `machine.config` as well as `web.config` within the `<configSections />` configuration element. By default a number of configuration section handlers are registered in the configuration files.

Because ASP.NET establishes the bin directory as one of the locations for resolving assemblies, it is possible to author configuration section handlers that reside within assemblies deployed to the bin directory. Because the application domain CAS policy has been set, any initialization logic that a user-authored configuration section handler executes when it loads is restricted to only those operations defined in the associated ASP.NET trust policy file. For example, in an ASP.NET application that runs at anything other than full trust, user code cannot call into Win32 APIs. As a result, in a partially trusted ASP.NET application, a web server

administrator is guaranteed that a malicious configuration section handler cannot make calls into Win32 APIs that attempt to reformat the hard drive (granted this is an extreme example, but you get the idea).

In Chapter 4 "Configuration System Security" the effects of ASP.NET trust levels on configuration will be discussed in more detail.

Establishing Assembly Locations

With the application domain policy set, ASP.NET performs some housekeeping that allows the .NET Framework assembly resolution to be aware of the bin directory. This allows the .NET Framework assembly resolution logic to probe the bin directory and resolve types from assemblies located within the "bin" directory. Remember that earlier ASP.NET performed some work to set up the code-generation directory structure. A side effect of this setup is that ASP.NET and the .NET Framework also have the ability to resolve types located in the application-specific code-generation directory.

ASP.NET also attempts to enable shadow copying of assemblies from the bin directory. Assuming that shadow copying is enabled, the .NET Framework will make private copies of these assemblies as necessary within the code-generation directory structure for the application. When the .NET Framework needs to reference types and code from assemblies in the bin directory, the framework will instead load information from the shadow copied versions. Shadow copying the bin assemblies allows you to copy new versions of assemblies into the bin directory without requiring the web application to be stopped. Because multiple web applications may be simultaneously running within a single worker process, the shadow copying behavior is important; it preserves the ability to maintain uptime for other web applications. If each application domain maintained a file lock on the assemblies located in the bin directory, XCOPY deployment of an ASP.NET application would be difficult. An administrator would have to cycle the entire worker process to release the file locks. With shadow copying, you can copy just new binaries to the server and ASP.NET will automatically handle shutting down the affected application domain and restarting it to pick up changes to the bin directory.

ASP.NET 2.0 introduces a new configuration element — `<hostingEnvironment />` — that administrators can used to disable shadow copying. The following configuration when placed within `<system.web />` will disabled shadow copying:

```
<hostingEnvironment shadowCopyBinAssemblies="false"/>
```

You may want to disable shadow copying if an administrator explicitly disallows overwriting assemblies on a production server. Disabling shadow copying would prevent someone from randomly updating an application's binaries when the application is already up and running. Also some assemblies expect that other files exist on the file system in the same directory structure as the assembly. In these cases, shadow copying causes the assembly to be shadow copied to a completely different directory structure, thus breaking the assembly's assumptions about relative file locations.

Obtaining the Auto-Generated Machine Key

If you have ever used viewstate or issued a forms authentication ticket, it is likely that you depended on an auto-generated machine key to provide security. The default `<machineKey />` configuration for an ASP.NET application sets both the `validationKey` and `decryptionKey` attributes to `AutoGenerate,IsolateApps`. During application domain initialization, ASP.NET ensures that the auto-generated machine key is available so that ASP.NET applications that depend on automatically generated keys will have the necessary key material.

The actual logic for generating and confirming the existence of the auto-generated machine key has changed over various versions of ASP.NET and with the different process models for hosting ASP.NET inside of IIS. Originally, when only Windows 2000 was available, the ASP.NET ISAPI extension would always run as SYSTEM because in IIS5 (and for that matter IIS 5.1), ISAPI filters and extensions always ran with the security credentials of the `inetinfo.exe` process. As a result, for IIS 5 and IIS 5.1, the ISAPI extension checks for the existence of the machine-generated key inside of the Local Security Authority (LSA). Because SYSTEM is such a highly privileged account, the ISAPI extension could safely generate and store the auto-generated machine key in the LSA.

However, with the process model in IIS6, ISAPI filters and extensions execute in a specific worker process. By default, the `w3wp.exe` worker process runs as NETWORK SERVICE, which has much fewer privileges than SYSTEM. As a result, the approach of storing items in LSA no longer works because NETWORK SERVICE does not have permission to read and write the LSA. Trust me when I say that this is a good thing (the idea of having your web server happily stuffing secret keys into the LSA is a little bit odd to say the least).

In IIS6, when running as NETWORK SERVICE the ASP.NET2.0 ISAPI extension will store and retrieve the auto-generated machine key from the following location in the Registry:

```
HKU\SID\Software\Microsoft\ASP.NET\2.0.50727.0
```

The value for the security identifier (SID) will vary depending on the identity of the worker process account. By default though when an IIS6 worker process runs as NETWORK SERVICE the SID will be `S-1-5-20`. Underneath this key are three values:

- **AutoGenKey** — This is the auto-generated machine key that is used for encryption and validation by forms authentication and for viewstate.
- **AutoGenKeyCreationTime** — An encoded representation of the file time when the key was generated.
- **AutoGenKeyFormat** — Indicates whether the auto-generated machine key was stored in an encrypted form (1) or as cleartext (2).

The very first time the ISAPI extension attempts to retrieve the auto-generated machine key, ASP.NET creates a random value, encrypts it using DPAPI (the extension uses the DPAPI user store), and stores the resultant information under the HKCU key mentioned earlier. In Figure 1-7, the auto-generated machine key information is stored in the user hive for NETWORK SERVICE. The SID `S-1-5-20` is the common SID representation for NETWORK SERVICE.

However, the question arises as to how the ISAPI extension can obtain an auto-generated machine key if the ASP.NET application is running as an account other than NETWORK SERVICE. For example, in IIS6 administrators commonly change the worker process identity to that of a local machine account or a domain account. Also, some web applications will use the `<identity />` element to configure a specific application impersonation identity.

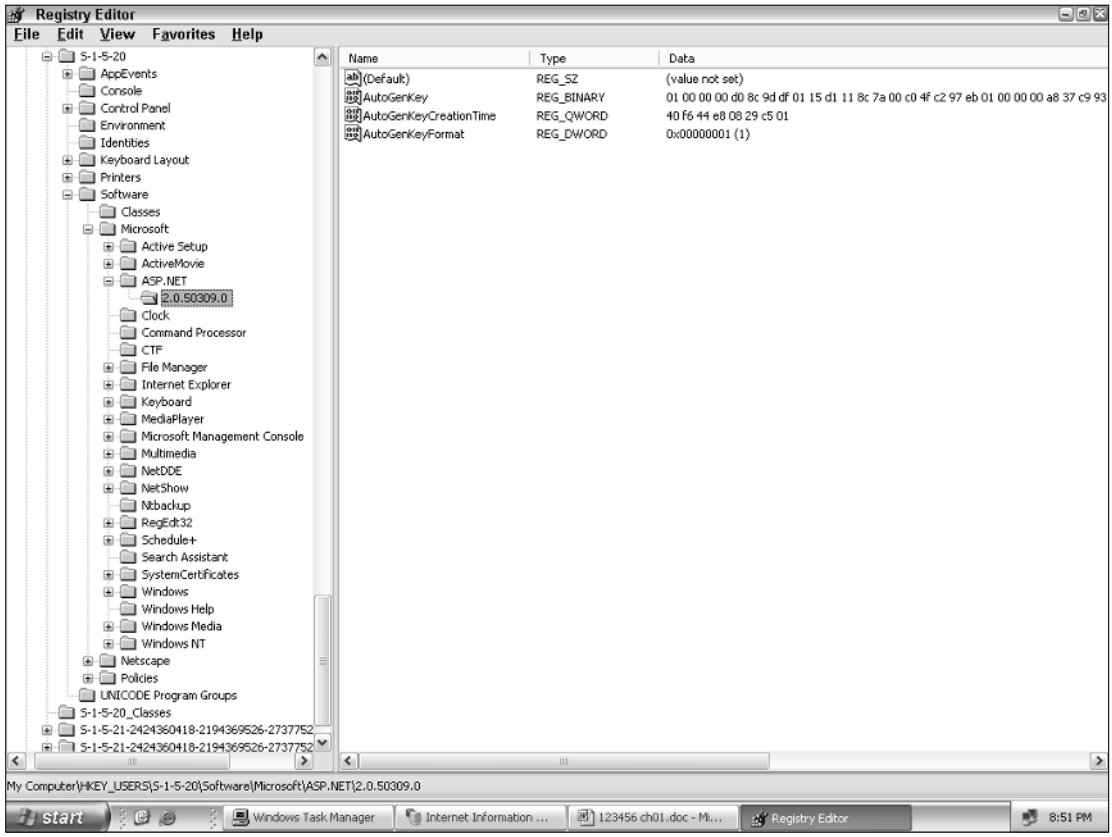

Figure 1-7

Although NETWORK SERVICE can store and retrieve the auto-generated machine key inside of the `HKEY_USERS` (HKU) area of the Registry, this technique will not work for local or domain accounts because accessing HKU requires that a user profile be loaded. Loading a user profile includes loading the portion of the Registry hive that is unique to a specific user. However, with IIS6 and ASP.NET, the user profile is loaded under only the following scenarios:

- The worker process is running as either NETWORK SERVICE or as LOCAL SERVICE.
- IIS6 is running in IIS5 isolation mode, in which case the user profile for the local ASPNET machine account will be loaded.

Other local and domain accounts *will not* have a user profile loaded on their behalf. As a result, ASP.NET needs some other location for storing the auto-generated machine key. If you choose to run ASP.NET with either a local or domain machine account, *always* make sure to run the following command line from the framework installation directory:

```
aspnet_regiis -ga DOMAIN\USERNAME
```

Running `aspnet_regiis` with the `ga` switch ensures that the ACLs for a variety of ASP.NET directories (remember the Temporary ASP.NET Files directory discussed earlier?) as well as ACLs in the IIS metabase are configured properly to grant access to the desired user account. Another side effect of using the `ga` switch though is that ASP.NET will create an `AutoGenKeys` Registry key at the following Registry location:

```
HKLM\SOFTWARE\Microsoft\ASP.NET\2.0.50727.0\AutoGenKeys
```

Underneath the `AutoGenKeys` key, the utility creates an additional key for the SID that corresponds to the user account that is currently being configured with the `ga` switch. This additional key will grant read and write access to the user account. As an example, Figure 1-8 shows the Registry location where `AutoGenKeys` has already been created. The only SIDs currently displayed in Figure 1-8 correspond to LOCAL SERVICE and NETWORK SERVICE and respectively. However, because the user profiles can be loaded for both of these accounts, no key information has been stored in the Registry.

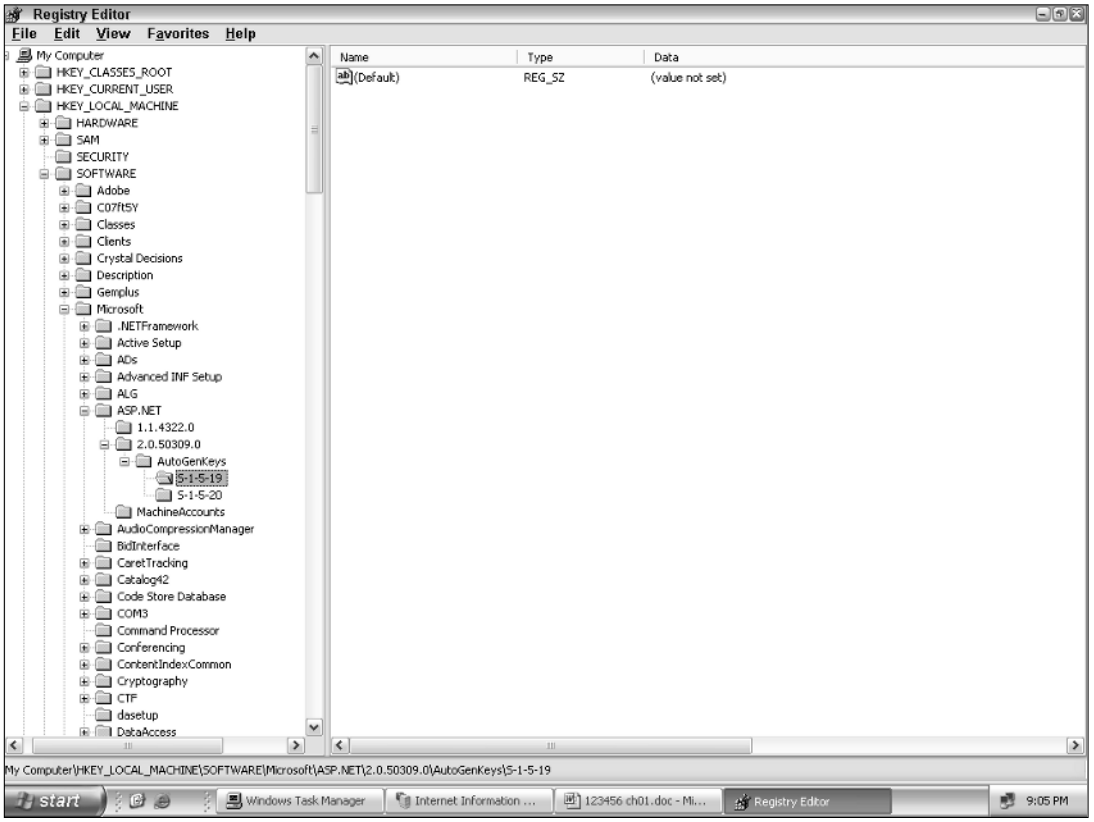

Figure 1-8

Assuming `aspnet_regiis -ga` has been used, when the ISAPI extension is initializing the application domain and is running as either a local or domain account, it will use neither LSA nor HKU and will instead create and access the auto-generated machine key information underneath:

```
HKLM\SOFTWARE\Microsoft\ASP.NET\2.0.50727.0\AutoGenKeys\SID
```

From all of this discussion, it should also be a bit clearer why using an auto-generated machine key in a web farm doesn't work. Regardless of which account is used for an ASP.NET application, the auto-generated machine key is local to a specific machine and furthermore to a specific user identity. As a result, applications running in a web farm (or in the case of forms authentication, applications running under different identities that need to recognize a common forms authentication ticket) must use explicit values for the `validationKey` and `decryptionKey` attributes in the `<machineKey />` configuration element. Explicit key values are the only way in ASP.NET 2.0 to ensure that the same keys are deployed on different machines. The DPAPI feature does not support exporting key material from one machine to another, so you don't have the option in a web farm of using the `AutoGenerate` setting. Realistically, configuring either of these attributes with `AutoGenerate` is only useful for smaller applications that can afford to run as standalone black boxes.

Initializing the Compilation System

During the last steps of application domain initialization, ASP.NET 2.0 initializes various aspects of its compilation system.

ASP.NET registers a custom assembly resolver to handle type load failures that arise when the .NET Framework cannot load a type that was defined in the `App_Code` directory. Code in the `App_Code` directory is compiled into in an auto-generated assembly that is assigned a random name. Each time a developer changes a piece of code that lives within the `App_Code` directory, ASP.NET will recompile the `App_Code` directory, which results in one or more new assemblies with different names (there can be subdirectories in `App_Code` that in turn give rise to multiple assemblies). As a result any operations that depended on the assembly name for a class located in `App_Code` (binary serialization for instance will write out the name of the assembly containing the serialized type) would fail without the ASP.NET custom assembly resolver. The resolver redirects requests for types from `App_Code` related assemblies to the most current versions of these auto-generated assemblies.

The ASP.NET runtime then ensures that various globally referenced assemblies are compiled and available. This includes ensuring the auto-compiled output for `App_Code`, the global and local resource directories, the `app_webreferences` directory and `global.asax` are up to date. As part of this processing, ASP.NET also starts file monitoring on `global.asax`. If any changes subsequently occur to `global.asax`, the changes cause the application domain to recycle.

First Request Initialization

With the application domain up and running, ASP.NET performs some initializations that occur only during the first request to the application domain. In general, these one-time tasks include the following:

- Caching the impersonation information so that ASP.NET knows the impersonation mode that is in effect for the application, as well as caching security tokens if application impersonation is being used or if the application is running on a UNC share.
- Configuration settings from `<httpRuntime />`, `<globalization />`, and `<processModel />` are loaded. The interesting point here is that you can use the `<httpRuntime />` element to turn off a website.

- A check is made to see if `App_Offline.htm` exists in the root of the website. If it does exist, requests are not served by the website
- The internal thread pools used by ASP.NET are set up based upon either the settings in configuration or using an heuristic if auto-configuration of thread settings was selected.
- Diagnostic and health related features are initialized. For example, ASP.NET initializes the counters for tracking the maximum number of queued requests as well as detecting that a response has deadlocked or hung. Part of this initialization also includes initializing tracing (as configured in `<trace />`) as well as starting the Health Monitoring feature (as configured in `<healthMonitoring />`).
- The compiled type for `global.asax` is loaded, and if `Application_Start` is defined in `global.asax`, it is called.

As you can see from this list, much of the work that occurs is internal and focused around initializing the internal workings of the ASP.NET runtime. However, a few steps are of interest from a security perspective and are discussed in more detail in the following sections.

Disabling a Website with the HttpRuntime Section

In ASP.NET 2.0, the `<httpRuntime />` configuration section has an enable attribute `\"`. By default it is set to `true`, but you can set the attribute to `false` as shown here:

```
<httpRuntime enable="false" />
```

Doing so causes ASP.NET to reject all requests made to the ASP.NET application. Instead of running the requested page (or handler), ASP.NET instead returns a 404 error indicating that the requested resource is not available. This setting is a pretty handy way to force an ASP.NET site to act as if it is offline while an administrator uploads new content or is making other modifications to a production web server.

Note that if you change this configuration setting on a live web server, the underlying application domain will restart because the configuration file changed.

Disabling a Website with App_Offline.htm

This is an alternative technique for indicating that an ASP.NET application is unavailable. If a file called `App_Offline.htm` is placed in the root of your website, all requests to the site return the contents of `App_Offline.htm` instead of running the requested page. Because it is an HTML file, you can place any static content you want into the file, and ASP.NET will stream it back to the browser. The one restriction is that the amount of content cannot exceed one megabyte. Of course, it is pretty unlikely that a developer would ever want to stuff that much content onto a page indicating that the site is unavailable.

As with the `enable` attribute of `<httpRuntime />`, placing `App_Offline.htm` into the root of your website causes the application domain to recycle. Additionally, when you remove the file from the root of your website, the application domain will recycle a second time. ASP.NET always has a file change monitor listening for this file so that it knows to recycle the application domain when the file's presence changes. The application domain recycling occurs only when the existence of `App_Offline.htm` changes. For example, after the file exists, there is an application domain up and running with the sole purpose of returning back the contents of the file. The application domain won't recycle again until the `App_Offline.htm` file is removed (or edited).

The Origins of App_Offline.htm

If you are wondering where the idea for `App_Offline.htm` originated, the idea was actually developed to handle a problem having nothing to do with security or website operations. SQL Server 2005 Express ships with the various versions of Visual Studio and includes a special mode of operation called user instancing. A side effect of user instancing is that SQL Server will hold a lock on your MDF database files while an ASP.NET application is accessing them. In production, of course, this isn't a problem. However, if you are developing against IIS using Visual Studio, and you frequently use Alt+Tab to switch between the website and the development tool, you would quickly run into problems trying to edit data in your database using Visual Studio. Hence the idea for `App_Offline`.htm.

Now when a developer attempts to edit data in the Visual Studio data designers, Visual Studio will first drop an `App_Offline.htm` file into the ASP.NET application's directory root. This has the effect of shutting down the ASP.NET application which in turn causes all outstanding ADO.NET connections to SQL Server Express 2005 to be released. As a result of the released connections, SQL Server Express 2005 detaches the MDF files thus making them available to be re-attached by the Visual Studio design time.

The advantage of using `App_Offline.htm` over the `<httpRuntime />` section though is twofold:

- It is trivial to automate usage of `App_Offline.htm`. Because it is just a file, administrative batch jobs or administrative tools do not need to write code to bring an ASP.NET application offline and then back online. As long as your administrative tools for your production servers can copy files, you can use the `App_Offline.htm` technique.
- You have easy control over the content that is sent back to your website users. With `<httpRuntime />`, the default content is generated by ASP.NET. In the case that your website disables remote error information with `<customErrors />`, you may have some control over error content assuming that you configured a custom error page for 404 errors. However, even if you use custom error pages, there is no way to distinguish between a 404 triggered by nonexistent website content, versus the 404 that ASP.NET generates when the application is offline. With `App_Offline.htm` you can create content for display to your users knowing that the information will be displayed only when the ASP.NET application has been taken offline.

Calling Application_Start in global.asax

Probably the most relevant startup activity for ASP.NET developers is the `Application_Start` event that can be authored in `global.asax`. Probably most developers that use `Application_Start` just breeze through writing the necessary code without worrying about the security context of this event. However, ASP.NET carefully manages the security context that is used to execute `Application_Start`.

Because the `Application_Start` event is written with user code, and the trust level has been previously established for the application domain, any code in the `Application_Start` event will be restricted based on the ASP.NET trust policy that was loaded for the application. Because the application domain initialization process also establishes a specific security identity, ASP.NET explicitly chooses an identity prior to running any code in the `Application_Start` event.

For example, one question that arises when running `global.asax` is what happens if client impersonation is in effect? To help frame this security problem, first a few terms should be discussed because using the shorthand for security contexts in ASP.NET is a lot faster than always calling out the `<identity />` element and its settings.

Client impersonation means that all of the following are true:

- Integrated Windows Authentication, Digest Authentication, Basic Authentication or some type of Certificate Mapping is configured for the ASP.NET application.
- The ASP.NET application's `<authentication />` element has the mode attribute set to `Windows`.
- The ASP.NET application's `<identity />` element has the impersonate attribute set to `true`.
- The ASP.NET application's `<identity />` element does not have the username or password attributes set.

An example of configuration settings that correspond to client impersonation is:

```
<identity impersonate="true" />
<authentication mode="Windows"/>
```

Application impersonation means that all of the following are true:

- The ASP.NET application's `<identity />` element has the impersonate attribute set to `true`.
- The ASP.NET application's `<identity />` element explicitly sets the values for the username and password attributes.

The value of `<authentication />` does not have any bearing on whether application impersonation is in effect. Within ASP.NET, code paths that look for the application impersonation identity will ignore any client credentials when an explicit application impersonation identity has been configured.

An example of configuration settings that correspond to application impersonation is:

```
<identity impersonate="true" userName="appimpersonation@corsair.com"
password="pass!word1"/>
```

UNC identity means that the ASP.NET application content is deployed remotely on a UNC share. When you configure an application to run on a UNC share in IIS, the IIS MMC prompts you to specify the way to handle credentials for the UNC share. In most web server environments an administrator supplies a unique username and password that have been granted read access to the remote share.

So, how does this all affect `Application_Start`? The underlying thread identity that ASP.NET uses when running `Application_Start` can only be that of the process identity, application impersonation identity, or the UNC identity. If client impersonation has been configured for an application, it is ignored while the `Application_Start` event is executing. This makes sense because if client impersonation were honored during `Application_Start`, you would end up with completely random behavior for any security-dependent operations running inside of the event. For example, if the client credentials were honored and a domain administrator just happened to be the first user that triggered application domain startup, everything might work properly. Yet if the website was recycled in the middle of the

day and the first person in afterwards had lower network privileges, then code inside of `Application_Start` would mysteriously fail. Limiting the security decision to one of process, application impersonation, or UNC identity guarantees stable security credentials each and every time the application starts up.

To highlight this behavior, use a simple ASP.NET application that stores the thread identity when `Application_Start` is running and then compares it to the thread identity that is used during a normal page request.

The sample application here uses the following code in `global.asax` to store the name of the authenticated identity that is used when `Application_Start` is called:

```
void Application_Start(Object sender, EventArgs e) {
    Application["WindowsIdentity"] =
        System.Security.Principal.WindowsIdentity.GetCurrent().Name;
}
```

You can then see the differences between the `Application_Start` identity and the actual identity that is running for a page request with the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Write("The operating system thread in Application_Start ran as: "
       + Application["WindowsIdentity"] + "<br/>");

    Response.Write("The current operating system thread identity is: " +
        System.Security.Principal.WindowsIdentity.GetCurrent().Name);
}
```

To see the effects of this, the code was run using a local ASP.NET application as well as a separate copy running remotely from a UNC share. The values for `<identity />` were varied as well, although in all cases Windows authentication was enabled for the application. The results of running the sample application in various configurations are shown in the following table:

Configured Impersonation	Running on UNC Share	Application_Start Thread Identity
None	No	NT AUTHORITY\NETWORK SERVICE
Client	No	NT AUTHORITY\NETWORK SERVICE
Application	No	The username as configured in `<identity />`
None	Yes	The UNC identity as configured in the IIS MMC
Client	Yes	The UNC identity as configured in the IIS MMC
Application	Yes	The username as configured in `<identity />`

The results for the non-UNC application make sense: Either the process identity or the application impersonation identity is used. The UNC case is a little bit trickier, because using application impersonation with a UNC share means that two sets of explicit credentials are floating around and being used by ASP.NET. When running as the application impersonation identity, some additional rights are needed for the application to run properly. The special security configurations need to fully enable UNC support as shown in the following table:

Configured Impersonation	Extra Security Configuration
None or Client	Because application initialization runs as the configured UNC identity, the UNC identity requires Modify access to the Temporary ASP.NET Files directory. However, it is also highly recommended that you configure the UNC identity with `aspnet_regiis -ga <UNC identity>`.
Application	Even though the application is on a UNC share, it is the application impersonation identity that is used to monitor change notifications for content files such as `global.asax` (recall the earlier discussion that described which identity is in effect during application domain initialization). As a result, the application impersonation identity requires read permissions on the UNC share (both share permissions and NTFS permissions).

If you plan to use code in `Application_Start` that depends on the security credentials associated with the operating system thread, you need to ensure that depending on how your application is configured the correct identity has rights to your backend data stores. For example, if you are planning on connecting to a database to fetch a dataset inside `Application_Start`, and you use Integrated Security with SQL Server; then the process identity, application impersonation identity, or the configured UNC identity need the appropriate rights on your SQL Server. The first two credentials make sense, but the UNC identity probably would catch some folks by surprise, especially if an application that was working fine when running from a local hard drive on a web server was moved to a UNC share on a production server. The moral of the story is that when running with a UNC identity, be careful and to test your application in an environment that closely mirrors the UNC structure you use in production.

> *Although the previous discussion centered on the* `Application_Start` *event, the same rules and rationale for determining security credentials are used when the* `Application_End` *event executes.*

Summary

In this chapter, you walked through many of the behind-the-scenes steps that occur when an application domain is started, as well as when the first request to the application domain is processed. Before a request is "seen" by the ASP.NET runtime though, the following hurdles must be cleared:

1. `http.sys` must consider the request to be well formed prior to passing it on to IIS
2. The ISAPI filter `aspnet_filter.dll` disallows any requests to special ASP.NET directories (`/bin`, `App_Data`, sand so on).
3. IIS determines whether the request is for static content or dynamic content. If IIS recognizes that the file extension for the requested resource is one that is mapped to ASP.NET, IIS forwards the request to ASP.NET's ISAPI extension
4. The ASP.NET ISAPI extension must complete a long series of steps that ultimately result in an application domain being spun up in-memory and prepared for executing ASP.NET requests

After the application domain is up and running, ASP.NET performs a few last steps for the very first request that is made to an application.

If you choose to run ASP.NET using local or domain accounts, make sure to run the `aspnet_regiis` utility with the `-ga` switch. Doing so will ensure that the necessary security rights have been granted and other setup tasks performed for these accounts to work properly.

Throughout all of the ASP.NET processing, the two most important security concepts to keep in mind are:

- ASP.NET configures and enforces an application domain CAS policy very early in the application domain's lifecycle. This means any code you write and deploy will be subject to the restrictions defined in an ASP.NET trust policy.
- The security credential that is used during application domain startup and during the early parts of the first request is one of the following: process identity, application impersonation identity, or UNC identity. Developers should understand which one is selected because code that runs during `Application_Start` uses one of these three identities.

The next chapter continues this discussion with a look at how the security context is set up for each individual request, as well as how the default handler mappings in ASP.NET provide security.