Security Extensibility in ASP.NET 4

By Stefan Schackow, ASP.NET team

This whitepaper covers the major ways in which security features in ASP.NET 4 can be customized, including:

- Encryption options and functionality in the `<machineKey>` element.
- Interoperability of ASP.NET 4 forms authentication tickets with ASP.NET 2.0.
- Configuration options to relax automatic security checks on inbound URLs.
- Pluggable request validation.
- Pluggable encoding for HTML elements, HTML attributes, HTTP headers, and URLs.

Table of Contents

Introduction

ASP.NET 2 introduced security features like user and role stores that integrated with forms authentication and URL authorization. In ASP.NET 4, the security infrastructure was updated, with the primary focus on adding extensibility points and configuration options that enable you to customize various security behaviors in ASP.NET.

This whitepaper does not cover code access security (CAS). For more details about that, see Code Access Security in ASP.NET 4 Applications on the MSDN website.

The <machineKey> Element and the MachineKey Utility Class

As with previous releases, ASP.NET 4 uses encryption and hashing information in the `<machineKey>` element to encrypt and sign cookies, view state, script resource URL, and membership passwords.

The <machineKey> Section

ASP.NET 4 introduced the following updates to functionality in the `<machineKey>` element of the configuration file:

- Support for keyed SHA-2 hash algorithms.
- Support for specifying custom classes.
- FIPS compliance.

Support for keyed SHA-2 hash algorithms

ASP.NET 4 includes support for keyed SHA-2 hash algorithms, defined using the `validation` attribute. You can also choose HMACSHA256, HMACSHA384 and HMACSHA512 as values.

> **Note** By default, ASP.NET 4 uses the HMACSHA256 keyed hash algorithm when it hashes items like cookies and view state. You may run into compatibility problems when you try to use features like forms authentication and try to share cookies across ASP.NET 2.0 and ASP.NET 4. See the next section for details.

Support for Specifying Custom Classes

Both the encryption and hash algorithm settings in `<machineKey>` can also be specified using a string identifier that represents a compatible cryptographic implementation. For the `validation` attribute, ASP.NET uses the string identifier from the configuration setting and calls `KeyedHashAlgorithm.Create(string)`. For the `decryption` attribute, ASP.NET uses the string identifier from the configuration setting and calls `SymmetricAlgorithm.Create(string)`.

As an example, the `validation` attribute of the `<machineKey>` element could specify a custom keyed hash algorithm like this:

```
<machineKey validation="alg:HMACSHA1" />
```

The `alg:` prefix tells ASP.NET that the remainder of the string value for the `validation` attribute represents the name of a keyed hash algorithm — in this case the `HMACSHA1` class.

Similarly, the `decryption` attribute of `<machineKey>` could specify a custom symmetric encryption algorithm like this:

```
<machineKey decryption="alg:AES" />
```

Again, the `alg:` prefix tells ASP.NET that the remainder of the string value for the `decryption` attribute represents the name of a symmetric encryption algorithm (here, the `AesCryptoServiceProvider` class).

FIPS Compliance

In ASP.NET 4 you can use the default `<machineKey>` settings, and all affected ASP.NET functionality will use FIPS-compliant algorithms without any additional configuration.

In previous versions of ASP.NET, the default encryption algorithm set in `<machineKey>` was AES. However, in ASP.NET 2.0 the underlying implementation was not FIPS-compliant, which required downgrading sites to 3DES if they required FIPS compliance. In ASP.NET 4, the internal implementation of AES has been changed to use the underlying operating system's native AES implementation. Therefore, AES can be used in ASP.NET 4 applications that require FIPS compliance. All of the new SHA-2 variants supported for hashing in ASP.NET 4 are FIPS compliant.

You can run sites where the configuration includes `<compilation debug="true" />` with FIPS compliance enabled. Internally, ASP.NET changed its usage of the MD5 algorithm to use a FIPS-compliant implementation. However, this means only *running* a website with FIPS compliance enabled will work in debug mode. Development-time activities involve more than just the ASP.NET runtime. Since Visual Studio 2010 itself is not FIPS compliant, IDE-related debugging tasks will still fail on machines where FIPS is enabled.

For more about FIPS compliance, see FIPS Compliance on the TechNet website.

The MachineKey Utility Class

ASP.NET internally performs all of the work for hashing and encryption operations based on settings in the `<machineKey>` section. However, a frequent developer request has been to expose these capabilities through public APIs. Although it's possible to use types like `FormsAuthentication` to accomplish this, before ASP.NET 4 there has not been a purpose-built type that gives you an easy-to-use API that uses ASP.NET hashing and encryption logic.

In ASP.NET 4, the `System.Web.Security.MachineKey` type was included expressly for this purpose. The `MachineKey` class exposes an API of two methods, an encoding method for hashing and encryption operations, and a decoding method for decrypting and verifying hash signatures.

The syntax for the new encoding method is this:

```
string Encode(byte[] data, MachineKeyProtection protectionOption)
```

You provide a byte array that represents the data to encode and an enumeration value indicating what operations should be performed on the contents of the byte array (keyed-hashing only, encryption only, or both). The `MachineKeyProtection` enumeration options mirror the options you have had for many years when encoding other pieces of ASP.NET state like forms authentication cookies.

When this method is invoked, ASP.NET 4 performs the requested operations against the byte array and returns the encoded results as a hexadecimal string. A hexadecimal string is returned because the primary intent of `<machineKey>` has been to encode data so it can safely be sent to a browser and eventually returned to the server via HTTP.

The syntax for the new decoding method is this:

```
byte[] Decode(string encodedData, MachineKeyProtection protectionOption)
```

You provide the encoded string that resulted from an earlier call to `Encode` and an enumeration value that indicates the operation (keyed-hash-verification only, decryption only, or both). ASP.NET 4 performs the requested operation. Assuming that the keyed hash signature is valid and that the decryption operation succeeds, ASP.NET returns the original byte array value.

A simple example of using the `MachineKey` class is shown below.

```
string value = "The quick brown fox jumped over the lazy dogs";

// Convert the Unicode string into a byte[] that can be encrypted
byte[] encodedValue = System.Text.Encoding.Unicode.GetBytes(value);

// Secure the byte array by both appending a keyed-hash to it,
// as well as encrypting the entire payload (both the string and the keyed-hash).
string hexString = System.Web.Security.MachineKey.Encode(
                       encodedValue,
                       System.Web.Security.MachineKeyProtection.All);

// Given the hex string output of a call to Encode, decrypt it and
// verify its keyed-hash signature.
byte[] decryptedBytes = System.Web.Security.MachineKey.Decode(
                       hexString,
                       System.Web.Security.MachineKeyProtection.All);

// Since the original value was a string, convert the byte array back
// into a Unicode string.
string finalValue = System.Text.Encoding.Unicode.GetString(decryptedBytes);
```

Note ASP.NET 4.5 includes two new methods that perform encoding/hashing functions, which will supersede these methods. For more information, see the **Protect** and **Unprotect** methods of the .NET 4.5 version of the MachineKey class.

Using the ASP.NET 4 <machineKey> Options with Membership

The `<machineKey>` section also interacts with the ASP.NET membership feature. Starting with ASP.NET 2.0, the membership feature used the `<machineKey>` information in two ways. By default, the membership feature hashes passwords with a random salt value, using the hash algorithm specified by the `<machineKey>` element's `validation` attribute. Effectively this means that by default, membership both hashes and salts passwords using the SHA1 algorithm. If you have changed the membership feature to instead use encrypted passwords, the encryption algorithm and key material are determined from the `<machineKey>` element's `decryption` and `decryptionKey` attributes.

ASP.NET 4 supports updated encryption and hashing algorithms, but the membership feature relies on persisted passwords. Therefore, by default the membership providers are restricted to the hash and encryption options available in ASP.NET 2.0. Membership providers do not automatically pick up new hash and encryption options from `<machineKey>`. This ensures that previously encrypted or hashed passwords still work in ASP.NET 4. In other words, if you upgrade an ASP.NET 2.0 or 3.5 website to ASP.NET 4, membership providers will continue to use either SHA1 (hashed and salted passwords) or AES (encrypted passwords) by default, and all of the previously saved passwords will continue to work without any problems.

If you want to use the additional hash and encryption algorithms available in `<machineKey>`, you can configure individual providers to do so using the provider-level `passwordCompatMode` attribute. The `passwordCompatMode` attribute has two possible values: "`Framework20`" (the default) and "`Framework40`".

For example, the following configuration example tells the `SqlMembershipProvider` class to use encrypted passwords and to enable encryption algorithms that may have been specified in the `<machineKey>` section. The encryption algorithm will be determined by the `decryption` setting in `<machineKey>`, which in this case is the AES encryption algorithm implemented by the `AesCryptoServiceProvider` class.

```
<machineKey decryption="alg:AES" ... />

<membership defaultProvider="SqlProvider">
  <providers>
    <add   name="SqlProvider"
        type="System.Web.Security.SqlMembershipProvider, etc..."
        connectionStringName="LocalSqlServer"
        applicationName="myApplication"
        passwordFormat="Encrypted"
        passwordCompatMode="Framework40"
    />
  </providers>
</membership>
```

Using ASP.NET 4 Forms Authentication Tickets with ASP.NET 2.0

It's usually impractical to update all of your websites at once to a new version of the .NET Framework, so ASP.NET has always enabled current and previous versions of ASP.NET to share forms authentication tickets. ASP.NET 4 continues to support interoperability of forms authentication tickets, for both cookies and for the cookieless ticket value.

As with previous major ASP.NET releases, forms authentication in ASP.NET 4 by default does not create tickets that are compatible with ASP.NET 2.0. As noted earlier, the default keyed-hash algorithm in ASP.NET 4 is HMACSHA256. However, ASP.NET 2.0 uses HMACSHA1. As a result, tickets issued by default in ASP.NET 4 cannot be consumed by ASP.NET 2.0 and vice versa.

To enable interoperable forms authentication tickets, make these configuration modifications:

- Configure forms authentication in ASP.NET 4 to use the older HMACSHA1 implementation by modifying the `validation` attribute in the `<machineKey>` section.

- Explicitly create both hash keys and encryption keys, and explicitly specify these keys in both the ASP.NET 4 and ASP.NET 2.0 *Web.config* files. (This step is easy to forget for scenarios where all applications are running on a single machine and you may be relying on ASP.NET autogenerated key material. Autogenerated keys cannot be used when applications share forms authentication cookies between ASP.NET 4 and ASP.NET 2.0.)

Note that by changing an ASP.NET 4 website to use SHA1 for keyed hashing, the website no longer uses the stronger HMACSHA256 algorithm for all keyed hash operations performed by ASP.NET.

The `<machineKey>` entry for both the ASP.NET 4 website and the ASP.NET 2.0 website that need to share tickets looks like the following:

```
<machineKey  validation="SHA1" validationKey="explicit value here"
    decryption="AES" decryptionKey="explicit value here" />
```

You provide your own values for the `validationKey` and `decryptionKey` attributes. You also need to ensure that the *Web.config* files for both the ASP.NET 4 and ASP.NET 2.0 websites have the exact same settings for the `validation`, `validationKey`, `decryption`, and `decrypionKey` attributes. (For `validation`, SHA1 is already the default in ASP.NET 2.0.)

As a side note, you may notice that ASP.NET 4 forms authentication configuration adds an attribute named `ticketCompatibilityMode`. By default, `ticketCompatibilityMode` is set to "`Framework20`". This ensures that after you have configured `<machineKey>` appropriately, ASP.NET 4 and ASP.NET 2.0 websites can share forms authentication tickets. If you change `ticketCompatibilityMode` to "`Framework40`", it is no longer possible to share tickets between ASP.NET 4 and ASP.NET 2.0, regardless of `<machineKey>` settings. The "`Framework40`" setting causes ASP.NET 4 forms authentication implementation to switch to consistently using UTC date-times, which results in an incompatible serialization format for the resulting forms authentication tickets.

Configuring Security Checks for Request URLs

Earlier versions of ASP.NET performed a variety of security checks against the request URL. However, some checks were performed only in native code, some checks were hard-coded in managed code, and some were optional checks occurring in managed code. Therefore, it was difficult for you to customize these URL security checks.

ASP.NET 4 makes all security checks for incoming URLs configurable and customizable through a combination of managed configuration settings and extensibility points. The following table shows the set of checks that occur for incoming URLs and the configuration settings introduced in ASP.NET 4 in the `<httpRuntime>` section that control these checks.

Security check	How to set	Granularity
Check length of `Request.Path`	`maxUrlLength` attribute in `<httpRuntime>` Defaults to rejecting URLs where the path is longer than 260 characters. This value can be increased so that longer URLs can be accommodated, up to the limit either set in IIS or enforced by *http.sys*.	Can be configured at the application level and for individual virtual paths and pages.
Check length of the query-string portion of `Request.RawUrl`	`maxQueryStringLength` attribute in `<httpRuntime>` Defaults to rejecting URLs where the query-string portion of the URL is more than 2048 characters. This value can be increased so that longer query strings can be accommodated, up to the limit either set in IIS or enforced by *http.sys*.	Can be configured at the application level and for individual virtual paths and pages.

Scan `Request.Path` for characters that ASP.NET considers potentially invalid	`requestPathInvalidCharacters` attribute in `<httpRuntime>`
	
	Defaults to rejecting URLs where the path portion contains one or more of the following characters:
	
	`< > * % : & \ ?`
	
	In the `requestPathInvalidCharacters` attribute, these characters are enclosed in quotation marks.
	
	The character sequence "`..`" is not on the list. In ASP.NET 4, the security checks for the "`..`" sequence were removed because on production servers running IIS 6, IIS 7, or IIS 7.5, the "`..`" sequence is automatically processed and removed as part of URL canonicalization prior to the request being passed to ASP.NET.
	
	In practice you will probably never see a URL rejected that contains the "`\`" character in the path. This is because IIS 6 and IIS 7 automatically reverse "`\`" is to "`/`" during URL canonicalization.
	
	Security Note If you remove any of the
		Can be configured at the application level and for individual virtual paths and pages.

	default characters from the list, you *must* check your code and ensure they are not accidentally opening your application to file canonicalization or XSS attacks. You must mitigate against any of the following potential attack vectors: • "<" (XSS attack) • ">" (XSS attack) • "*" (File canonicalization attack) • "%" (Attack via URL decoding logic and URL assumptions) • ":" (Alternate NTFS data stream attack) • "&" (Attack via custom query-string parsing) • "?" (Attack via custom query-string parsing)	

Check `Request.Path`, `Request.PathInfo`, and `Request.RawUrl` for potential XSS (cross-site scripting) strings	`requestValidationMode` attribute in `<httpRuntime>`
	
	This attribute controls how soon request validation is enabled. By default, in ASP.NET 4 request validation occurs before any custom HTTP modules are called as part of `BeginRequest`. This attribute can be set to "`2.0`" to revert to ASP.NET 2.0 behavior.
	
	`requestValidationType` in `<httpRuntime>`
	
	By default, ASP.NET 4 performs the same request validation checks as in previous versions. However, you can plug in a custom request validation check, as explained under Changes to Request Validation later in this document.
		`requestValidationMode` can be configured at the application level and for individual virtual paths and pages.
		
		`requestValidationType` can be configured only in the application-level *Web.config* file.

Look up the appropriate managed configuration information for each value of `Request.Path`	`relaxedUrlToFileSystemMapping` attribute in `<httpRuntime>` By default, as part of the configuration lookup, ASP.NET assumes the path portion URL is a valid NTFS file path—that is, this attribute is set to false by default. You can disable this constraint by setting `relaxedUrlToFileSystemMapping` to true.	Can be configured only in the application-level *Web.config* file.

	If you disable this constraint, you should validate request URLs and constrain them to work only with URL patterns that the application expects. In other words, by disabling this check, you disable the implicit protection that is provided by ASP.NET where the configuration system enforced the constraint that request URLs are valid NTFS directory paths.	

The `requestValidationMode` and `requestValidationType` attributes are covered in more detail later in this document in the section on request validation. The following examples demonstrate how to use the other configuration settings.

Example 1: Long URLs

If a request uses a long URL, by default ASP.NET will reject it if the value of `Request.Path` exceeds 260 characters. For example, imagine a URL like this:

```
http://localhost/some/really/long/sequence/of/paths/etc/etc ...
```

(Assume that the URL goes on beyond 260 characters.)

ASP.NET will return an exception with the following text:

> The length of the URL for this request exceeds the configured maxUrlLength value.

You can configure ASP.NET to allow longer URLs by using the following setting:

```
<httpRuntime maxUrlLength="1024" ... />
```

However, after you make this change, you will encounter an exception with the following text:

> The specified path, file name, or both are too long. The fully qualified file name must be less than 260 characters, and the directory name must be less than 248 characters.

Furthermore, the stack trace will likely end somewhere in a call to a `System.IO` API, with a call further up the stack to `System.Web.HttpRequest.get_PhysicalPath`. This behavior occurs because even though the ASP.NET URL length check has been modified, ASP.NET still attempts to match the long URL to the correct managed configuration record. By default, this involves checking directory paths on the file system, and these file I/O calls fail because the supplied virtual path exceeds the NTFS directory length limits.

You can relax this requirement by setting `relaxedUrlToFileSystemMapping` , like this:

```
<httpRuntime relaxedUrlToFileSystemMapping="true" maxUrlLength="1024" ... />
```

With this addition, the request that uses the long URL will succeed. ASP.NET will ignore file I/O errors that occur during configuration mapping. Instead, it will internally truncate the inbound URL, path segment by path segment, until it finds a subset of the URL that can be successfully matched to a directory. In the extreme case, ASP.NET will eventually truncate the URL down to the application's root URL (just `/` or something like `/yourapproothere`). In that case, the application-level *Web.config* file is used to define the configuration settings that apply to the incoming URL. This truncation is an internal process that occurs during configuration mapping. It has no effect on the URL or path values that you retrieve from objects like `HttpRequest`.

As a side note, another way you will know that `relaxedUrlToFileSystemMapping` is needed (after you have seen exceptions) is if you display the value of `Request.PhysicalPath` from a web page. If ASP.NET has performed its fallback logic for matching configuration records, the value returned from `Request.PhysicalPath` will look something like the following:

```
C:\inetpub\wwwroot\myapplication\NOT_A_VALID_FILESYSTEM_PATH
```

The string constant `NOT_A_VALID_FILESYSTEM_PATH` tells you that the request URL is not a valid physical NTFS file path. The assumption here is that since an application was explicitly designed with long URLs in mind, it likely has the necessary logic (for example, URL routing) to parse the URL and process the request further without requiring a physical *.aspx* page that has a matching directory path and file name.

Example 2: URL with an Invalid Character

Websites frequently include data as part of path segments in a URL instead of as a value in a query string. This can be done to create "hackable" URLs that make it easier for users to navigate a website by modifying portions of a URL. Including data in path segments also enables websites to have URLs that are search-engine friendly, because search engines do not include query-string values when indexing a website.

Imagine you want to enable a URL like this:

```
http://localhost/myecommercesite/catalog/gardening|lawncare
```

In ASP.NET 4, you must relax the file-mapping behavior for configuration, because the pipe character ("`|`") is not allowed in NTFS file paths. Use a setting like this in the *Web.config* file:

```
<httpRuntime relaxedUrlToFileSystemMapping="true" ... />
```

Imagine further that you decide not to use the "`|`" character as a delimiter and instead use a colon ("`:`"), like this:

```
http://localhost/myecommercesite/catalog/gardening:lawncare
```

You must now relax both file-mapping behavior (":" is also invalid in NTFS file paths) and the default URL character checks, using settings like this:

```
<httpRuntime  requestPathInvalidCharacters="&lt;,&gt;,*,%,&amp;,\,?"
  relaxedUrlToFileSystemMapping="true" ... />
```

In this case, you are removing the ":" value from the list in `requestPathInvalidCharacters`. However by using the ":" character as a delimiter, you might have opened up an attack vector in your application. For example, a malicious user might try to send a URL like this one:

```
http://localhost/myecommercesite/catalog/gardening::$DATA
```

If the application code stripped the end of the URL and blindly used that as input to a file I/O API, the application could end up returning the alternate data stream of an NTFS file. (This was a well-publicized attack vector years ago with classic ASP.)

As noted earlier, although ASP.NET 4 enables you to handle a wider range of URLs, the potential security risks must be understood and mitigated in application code.

Example 3: URL with an XML Payload

Another variation of a non-NTFS URL is one that contains an XML element. For example, perhaps you want to construct URLs like this:

```
http://localhost/approot/subpath/<data>somedatahere</data>
```

By default, ASP.NET will return the following error:

```
A potentially dangerous Request.Path value was detected from the client (<).
```

This occurs because the "<" character is one of the characters that is automatically blocked, as defined by the `requestPathInvalidCharacters` attribute. Since the ">" character is also blocked by the `requestPathInvalidCharacters` setting, you need to remove both characters from the block list. The configuration example below shows both characters removed:

```
<httpRuntime requestPathInvalidCharacters="*,%,:,&amp;,\,?" />
```

However, after you make this change, ASP.NET 4 will still return a validation error, although it's a different one:

```
A potentially dangerous Request.Path value was detected from the client
(="../approot/subpath/<data>somedatahere</dat...").
```

This occurs because although the URL scanning check has been relaxed to allow "<" and ">" characters, ASP.NET request validation is also running and looking at the values of `Request.RawUrl`, `Request.Path`, and `Request.PathInfo`. The request validation check looks for strings that are often used in XSS attacks. As a result, request validation rejects URLs that seem to include script or HTML tags.

You can relax the request validation checks in different ways, depending on what technology is being used to process the request. For purposes of this example, assume you're using Web Forms and that page routing is being used to forward virtual URLs to *.aspx* pages. In that case, you need to carry out two additional steps.

First, you can turn off request validation for the specific *.aspx* page using the `ValidateRequest` attribute in the `@ Page` directive:

```
<%@ Page Language="C#" ...  ValidateRequest="false" ... %>
```

Since ASP.NET 4 is more aggressive with request validation than previous versions of ASP.NET (see also the next section of this document), this setting enables request validation so that it runs before any user code, during the `BeginRequest` phase of the HTTP pipeline. As a result, in order for the page-level `ValidateRequest` attribute to take effect, you also must tell ASP.NET 4 to not enable request validation before this stage of processing.

To do so, use the `requestValidationMode` attribute in the `httpRuntime` section. However, instead of downgrading request validation behavior for the entire application, you should only relax it for the specific path or paths that require it. The `<httpRuntime>` element to specify relaxed request validation looks like the following:

```
<location path="subpath">
  <system.web>
    <httpRuntime
      requestValidationMode="2.0"
      requestPathInvalidCharacters="*,%,:,&amp;amp;,\,?" />
    />
  </system.web>
</location>
```

In this example, the `<httpRuntime>` element is in a `<location>` element so that relaxed URL checks are applied only to a narrower set of URLs. As with any case where the default protections of ASP.NET are relaxed, you *must* make sure that the application includes custom validation logic is written to ensure that an XSS vulnerability has not been accidentally introduced.

If you're using ASP.NET MVC or ASP.NET Web Pages (*.cshtml* files), the method for disabling request validation is a little different.

Changes to Request Validation

Request validation was first introduced in ASP.NET 1.1 to provide basic protection against potential cross-site scripting (XSS) attacks. During validation, ASP.NET scans values from an HTTP request and throws an error if "potentially dangerous" values are found — that is, HTML tags, script snippets, and so

on. In subsequent versions of ASP.NET, small adjustments were made to expand the number of HTTP inputs that are checked and the types of patterns that are looked for.

In ASP.NET 4, request validation improvements fall into these categories:

- Request validation turns on earlier and checks more values by default.
- You can plug-in custom request validation logic.

What Request Validation Checks For

The following list summarizes what request validation looks for by default. (The descriptions are not highly precise.)

- The character "`<`" followed by any of these:
  - `a-z` or `A-Z`
  - `!`
  - `/`
  - `?`
- The character "`&`" followed by the character "`#`".
- The word "`script`" followed by a colon (`:`), potentially with whitespace between them.

These checks mean that ASP.NET rejects HTTP inputs that look like HTML tags (opening or closing), XML tags (opening or closing), XML comments, XML processing directives, and HTML entity references.

Changes to When Validation Runs

In previous versions of ASP.NET, request validation was invoked relatively late in the HTTP pipeline. For Web Forms pages, request validation was invoked when the *.aspx* page was initializing. This has meant that you were able to consume unvalidated data from an HTTP request before page execution (that is, before the handler-execute phase in the HTTP pipeline).

For example, an HTTP module that runs during the authenticate-request phase (before page execution) might read a cookie and store its value for later use when an *.aspx* page runs. If the HTTP module didn't perform its own validation checks on the cookie value, the page code could use this value even though the cookie value had an XSS string embedded in it.

The scenario is one that ASP.NET MVC ran into early in its development. MVC applications don't follow the execution pattern of*.aspx* pages. Instead, ASP.NET parses path segments of a URL and passing bits of the path segments into controller classes as either data in a route dictionary or as parameters to an action method. Because of this model, in MVC applications ASP.NET explicitly invokes request validation before it passes URL data to application code. If ASP.NET didn't do this, it would have been possible to embed XSS values into URLs and pass unvalidated values into application code.

We've learned that many developers weren't perfectly clear about when request validation was being invoked. In addition, in MVC and Web Pages applications, ASP.NET code doesn't run in an *.aspx* page. Because of this, ASP.NET 4 invokes request validation before any user code runs — during the `BeginRequest` phase of the HTTP pipeline. Basically, ASP.NET flips a number of internal flags that indicate request validation is enabled, and then ASP.NET 4 calls into the first HTTP module that is registered for the `BeginRequest` event (if any).

This change means that by default, the value of the `validateRequest` attribute of the `<pages>` element in the configuration file is ignored, as is the `ValidateRequest` attribute in the `@ Page` directive. This is because request validation will already have been invoked long before a page starts running. Similarly, by default the value of the `ValidateRequest` attribute on MVC controllers is ignored.

If it makes sense for your application, you can revert request validation to its ASP.NET 2.0 behavior. For example, you might do this if you already have individual *.aspx* pages whose `ValidateRequest` attribute in `@ Page` is set to false in order to allow users to post HTML markup. You might also revert to the earlier behavior for MVC controllers that use the `ValidateRequest` attribute to disable request validation. You have two ways to do this: you can revert the entire application to the ASP.NET 2.0 validation behavior, or you can selectively revert individual paths or pages to the older behavior.

The recommendation is to relax request validation only for the virtual paths or specific pages where this is needed. The following configuration example shows an example of reverting request validation to the ASP.NET 2.0 behavior, thus allowing the `ValidateRequest` setting on the *test.aspx* page to take effect:

```
<location path="test.aspx">
  <system.web>
    <httpRuntime requestValidationMode="2.0" />
  </system.web>
</location>
```

Changes to What Request Validation Checks

ASP.NET 4 also expands the set of HTTP inputs that are subject to request validation by including `Request.Path` and `Request.PathInfo` in all request validation checks. To summarize ASP.NET 4 request validation behavior, by default the following HTTP request values are automatically checked:

- All query-string values.
- The values of all form variables.
- The values of all cookies.
- The names of files (if any) contained in `Request.Files`.
- The values of `Request.RawUrl`, `Request.Path`, and `Request.PathInfo`.

In addition, ASP.NET 4 passes the raw values of all HTTP headers to any custom request validation implementations (see the next section), although by default ASP.NET itself does not perform any additional checks on those values.

Pluggable Request Validation

Efforts in the past to make the default request validation more restrictive invariably resulted in compatibility issues with some number of existing ASP.NET websites. Therefore, instead of trying to tune the strings that built-in request validation looks for, we decided on a different approach.

The big change for ASP.NET 4 is that request validation is pluggable. By default your application runs the set of checks as in earlier versions. However, in ASP.NET 4 you can supplement or entirely replace ASP.NET request validation logic with a custom implementation. For example, you can write a custom request validation implementation that scans for SQL injection attacks in addition to looking for XSS attacks.

Creating the Custom Request Validator

The first step to developing a custom request validation plugin is to write a custom class that derives from the base type `System.Web.Util.RequestValidator`. The `RequestValidator` type has the following signature:

```
public class RequestValidator
{
  public static RequestValidator Current {get;}
  protected virtual bool IsValidRequestString(
            HttpContext context,
            string value,
            RequestValidationSource requestValidationSource,
            string collectionKey,
            out int validationFailureIndex);
}
```

ASP.NET calls into a custom request validation implementation for each piece of HTTP request data that is validated. This means that each raw HTTP header and various `Request` properties and values described earlier are passed into a custom request validator.

The following table describes the parameters for the `IsValidRequestString` method.

Parameter	Description
`Context`	The `HttpContext` instance of the current request, which provides access to information about the current request. You typically check `context.Request.Url` or another URL-related property in a custom request validation implementation. Accessing a property like `context.Request.Url` triggers an additional (recursive) request validation check, because all URL properties are themselves subject to request validation. If you are validating query-string values, the custom request validator can potentially check these values: • `RawUrl`, since the query-string is a part of the overall raw URL. • Individual query-string values passed to the custom validator. If application code only needs to access query string values, it can use `Request.QueryString`, set `requestValidationSource` to `QueryString`, and ignore `RawUrl`. In that case the custom validator can ignore `RawURL`. The value for `Request.RawUrl` is used as the source for the values of the `Request.Path`, `Request.PathInfo`, and `Request.QueryString` properties. In your custom request validator, you must decide whether `RawUrl` should be included in any validation checks. For example, if you want to allow strings that contain might contain markup or code inside a path segment, you could write validation logic to examine `Path` and `PathInfo`. However, ASP.NET will still check `RawUrl` and will throw an exception if it encounters "dangerous" values in `RawUrl`. As a result you need to decide if explicitly ignoring `RawUrl` values in a custom validator is appropriate. Alternatively, you can use custom logic to check `RawUrl`, though `RawUrl` contains a pre-processed version of the original URL. For example, the

	query-string data in `RawUrl` is not in a parsed or URL-decoded format. When you write custom code to check `RawUrl`, you need to account for these differences.
`requestValidationSource`	An enumeration value that describes what type of HTTP input is being passed in `value`. This can be used as a way to quickly fall back to the base request validation implementation for pieces of HTTP input that you don't care about validating with custom logic.
`collectionKey`	The string key that identifies `value` if the HTTP input came from a collection. For example, if cookies are currently being checked, `collectionKey` is the name of an individual cookie, and `value` contains the cookie value. For single-valued HTTP inputs like `RawUrl`, this parameter is null.
`value`	The value of the HTTP input to validate.
`validationFailureIndex`	(out parameter) A zero-based index into `value` that indicates where custom request validation found a suspicious value. ASP.NET uses this index to create the request validation error message. If custom request validation does not find any suspicious values, this parameter should be set to -1.

The method should return `true` if the input passes validation — that is, if custom request validator did not find any suspicious strings. If the validator encounters something suspicious, it should return `false` and set the starting index of the suspicious string in `validationFailureIndex`.

The following example shows a simple custom validator that allows the query-string key named "data" to contain a specific XML element and value. This custom validator will allow a URL of the form `/appproot?data=<myTag>1234</myTag>`. However, it will reject a URL that has anything other than "`1234`" in the `<myTag>` element, and any other element. (For example, `/appproot?data=<myTag>other-value</myTag>`.) It will also reject a URL that has XML anywhere else in the query string.

```
using System;
using System.Web;
using System.Web.Util;

public class CustomRequestValidation : RequestValidator
{
    public CustomRequestValidation() {}
```

```
    protected override bool IsValidRequestString(
          HttpContext context, string value,
          RequestValidationSource requestValidationSource, string collectionKey,
          out int validationFailureIndex)
    {
        validationFailureIndex = -1;  //set a default value for the out param

        // This application doesn't use RawUrl directly, so ignoring
        // the check is okay.
        if (requestValidationSource == RequestValidationSource.RawUrl)
            return true;

        // Allow the query-string key named "data" to have an XML-like value
        if (
            (requestValidationSource == RequestValidationSource.QueryString) &&
            (collectionKey == "data")
           )
        {
            // The querystring value "<myTag>1234</myTag>" is allowed
            if (value == "<myTag>1234</myTag>")
            {
                validationFailureIndex = -1;
                return true;
            }
            else
            {
                // Delegate any further checks back to ASP.NET
                return base.IsValidRequestString(
                    context, value, requestValidationSource,
                    collectionKey, out validationFailureIndex);
            }
        }
        // All other HTTP input checks are delegated back
        // to the default ASP.NET implementation.
        else
        {
            return base.IsValidRequestString(
                    context, value, requestValidationSource,
                    collectionKey, out validationFailureIndex);
        }
    }
}
```

Configuring an ASP.NET Site to Use a Custom Validator

After you've written a custom request validator, you configure ASP.NET to use it with the following application-level *Web.config* file setting:

```
<httpRuntime requestValidationType="customRequestValidationType" />
```

As with other type attributes in configuration, `requestValidationType` is a standard .NET type string. The encoder type can be defined in the *App_Code* folder, in a class library that's in the *Bin* folder, or in a class library installed in the GAC.

Only one custom request validation type can be configured for an application — you can't configure a different request validation type for individual virtual paths or pages.

Pluggable Encoding for HTML, Attributes, HTTP Headers, and URLs

It's a security best practice for web development to encode outbound data in HTTP responses. However, even when a website is always encoding its data, differences can arise over how aggressive or relaxed an encoding algorithm should be.

Therefore, ASP.NET 4 makes its string encoding routines pluggable. You can write a custom encoding class and configure ASP.NET to use it to either replace or supplement the following encoding behaviors:

- HTML encoding
- HTML attribute encoding
- URL encoding
- URL path encoding
- HTTP header name and header value encoding

You might not be familiar with URL path encoding. This is the encoding algorithm applied by ASP.NET whenever `HttpServerUtility.UrlPathEncode` is called. Although you probably don't call `UrlPathEncode`, ASP.NET internally makes heavy use of this method when writing out HTML attributes such as the `action` attribute of a `<form>` tag or the `href` attribute on an `<a>` element.

`UrlPathEncode` automatically URL-encodes any values from 0x00 to 0x1F and all values from 0x7F and above. It is also encodes spaces into the corresponding `%20` sequence. However, since URL path segments are now often used to hold data, the current behavior of `UrlPathEncode` may be too simplistic, so you can extend the default encoding and write a more aggressive version of `UrlPathEncode`.

Another common encoding issue is that the default ASP.NET `UrlEncode` behavior converts spaces into the "+" character. This behavior is a historical holdover from when "+" was an Internet convention for

encoding space characters on the URL into a more human-readable fashion. However, from a standards perspective, the space character is supposed to be encoded as `%20` and not as a "`+`" character. The encoding extensibility in ASP.NET 4 makes it possible for you to implement a more standards-compliant encoding algorithm.

You write a custom encoder by deriving from the `System.Web.Util.HttpEncoder` type. The full type signature is shown below.

```
public class HttpEncoder
{
  // Returns the encoder that has been configured for the application.
  // This may be different than the default ASP.NET default implementation.
  public static HttpEncoder Current {get; set;}

  // Returns an encoder reference that is ASP.NET's default encoder logic.
  public static HttpEncoder Default {get;}

  protected internal virtual byte[] UrlEncode(byte[] bytes, int offset,
                             int count);
  protected internal virtual string UrlPathEncode(string value);

  protected internal virtual void HtmlEncode(string value, TextWriter output);
  protected internal virtual void HtmlDecode(string value, TextWriter output)

  protected internal virtual void HtmlAttributeEncode(string value,
                                  TextWriter output);

  protected internal virtual void HeaderNameValueEncode(
                             string headerName,
                             string headerValue,
                             out string encodedHeaderName,
                             out string encodedHeaderValue);
  }
}
```

You don't have to override every method. Because ASP.NET has default functionality for all of the virtual encoding methods, you can override just the method or methods you want to customize.

To configure ASP.NET to use your custom encoder, specify it in the `<httpRuntime>` section of the configuration file using the `encoderType` attribute:

```
<httpRuntime encoderType="MyCustomEncodeType" />
```

Only one custom encoding type can be configured per application.

The following example shows a custom encoder that simply routes HTML encoding calls into similarly named methods of the `Microsoft.Security.Application.AntiXSS` type in the Anti-XSS library.

Note The Anti-XSS library is integrated into the .NET Framework 4.5. In the integrated version of the library, you call methods of the `Encoder` type instead of calling methods of the `AntiXss` type.

```
using System;
using System.Web;
using System.Web.Util;
using Microsoft.Security.Application;

public class AntiXssEncoder : HttpEncoder
{
    public AntiXssEncoder() { }

    protected override void HtmlEncode(string value, System.IO.TextWriter output)
    {
        output.Write(AntiXss.HtmlEncode(value));
    }

    protected override void HtmlAttributeEncode(string value,
                            System.IO.TextWriter output)
    {
        output.Write(AntiXss.HtmlAttributeEncode(value));
    }
}
```

The following configuration example shows how to configure an ASP.NET application to use this encoder. In this example the custom encoder is assumed to be in the *App_Code* directory, so the `type` attribute is very simple:

```
<httpRuntime encoderType="AntiXssEncoder" />
```

Imagine that you have an *.aspx* page with the following markup:

```
 <form id="form1" runat="server">
    <div>
       <%-- Note the use of the new <%: syntax below to auto-HTML-encode
            the enclosed value --%>
       HTML encoded value:  <%: Request.QueryString["data"] %>
       <br/><br/>
       <asp:HyperLink ID="HyperLink1" runat="server">
            This is a hyperlink control
       </asp:HyperLink>
```

```
    </div>
</form>
```

A request URL like `http://localhost/yourapp/Default.aspx?data=123<4567` will cause ASP.NET to render the markup from the `<div>` element like this:

```
<div>
   HTML encoded value:  123&lt;4567
   <br/><br/>
   <a id="HyperLink1" href="123&lt;4567">
       This is a hyperlink control
   </a>
   <br/>
</div>
```

The `<%: Request.QueryString["data"] %>` syntax automatically HTML-encodes the value that's inside the `<%: %>` tags. Since the custom encoder overrides the HTML encoding method, the resulting output shows how the data value "123<4567" was encoded by the Anti-XSS library.

When ASP.NET is rendering the hyperlink control, it HTML-encodes attributes and URL-encodes the hyperlink's `NavigateUrl` property. The page code for the previous example (not shown here) is assumed to set the hyperlink control's `NavigateUrl` property to the query-string value named "`data`". Because the custom encoder also overrides the `HtmlAttributeEncode` method, the resulting output shows how the "`<`" character is encoded in the `href` of the `<a>` element using the HTML entity "`<`".

A powerful feature of the custom encoding behavior is that you don't have to explicitly change any code for a page in order to use a different encoding library. By configuring ASP.NET to use a custom encoder, all of the encoding functionality within ASP.NET automatically uses the custom encoder.