Tell us about your PDF experience.

Configuring Internet Applications

Article • 03/30/2023

The <system.Net> Element (Network Settings) configuration element contains network configuration information for applications. Using the <system.Net> Element (Network Settings) element, you can set proxy servers, set connection management parameters, and include custom authentication and request modules in your application.

The <defaultProxy> Element (Network Settings) element defines the proxy server returned by the `GlobalProxySelection` class. Any HttpWebRequest that does not have its own Proxy property set to a specific value uses the default proxy. In addition to setting the proxy address, you can create a list of server addresses that will not use the proxy, and you can indicate that the proxy should not be used for local addresses.

It is important to note that system's Internet settings are combined with the configuration settings, with the latter taking precedence.

The following example sets the default proxy server address to `http://proxyserver`, indicates that the proxy should not be used for local addresses, and specifies that all requests to servers located in the contoso.com domain should bypass the proxy.

XML

```
<configuration>
    <system.net>
        <defaultProxy>
            <proxy
                usesystemdefault = "false"
                proxyaddress = "http://proxyserver:80"
                bypassonlocal = "true"
            />
            <bypasslist>
                <add address="http://[a-z]+\.contoso\.com/" />
            </bypasslist>
        </defaultProxy>
    </system.net>
</configuration>
```

Use the <connectionManagement> Element (Network Settings) element to configure the number of persistent connections that can be made to a specific server or to all other servers. The following example configures the application to use two persistent connections to the server `www.contoso.com`, four persistent connections to the server with the IP address 192.168.1.2, and one persistent connection to all other servers.

XML

```
<configuration>
    <system.net>
        <connectionManagement>
            <add address="http://www.contoso.com" maxconnection="2" />
            <add address="192.168.1.2" maxconnection="4" />
            <add address="*" maxconnection="1" />
        </connectionManagement>
    </system.net>
</configuration>
```

Custom authentication modules are configured with the <authenticationModules> Element (Network Settings) element. Custom authentication modules must implement the IAuthenticationModule interface.

The following example configures a custom authentication module.

XML

```
<configuration>
    <system.net>
        <authenticationModules>
            <add type="MyAuthModule, MyAuthModule.dll" />
        </authenticationModules>
    </system.net>
</configuration>
```

You can use the <webRequestModules> Element (Network Settings) element to configure your application to use custom protocol-specific modules to request information from Internet resources. The specified modules must implement the IWebRequestCreate interface. You can override the default HTTP, HTTPS, and file request modules by specifying your custom module in the configuration file, as in the following example.

XML

```
<configuration>
    <system.net>
        <webRequestModules>
            <add
                prefix="HTTP"
                type = "MyHttpRequest.dll, MyHttpRequestCreator"
            />
        </webRequestModules>
    </system.net>
</configuration>
```

See also

- Network programming in .NET
- Network Settings Schema
- <system.Net> Element (Network Settings)

Network Tracing in the .NET Framework

Article • 09/15/2021

Network tracing in the .NET Framework provides access to information about method invocations and network traffic generated by a managed application. This feature is useful for debugging applications under development as well as for analyzing deployed applications. The output provided by network tracing is customizable to support different usage scenarios at development time and in a production environment.

To enable network tracing in the .NET Framework, you must select a destination for tracing output and add network tracing configuration settings to either the application or machine configuration file. For descriptions of configuration files and how they are used, see Configuration Files. For information about how to enable network tracing, see Enabling Network Tracing. For information about the settings that you need to add to the configuration file, see How to: Configure Network Tracing.

When tracing is enabled, you can capture trace information that is output by **System.Net** classes. Networking class members that generate tracing information include the following note in the Remarks section of their NET Framework class library documentation:

> ⓘ **Note**
>
> This member outputs trace information when you enable network tracing in your application. For more information, see Network Tracing.

See also

- Enabling Network Tracing
- How to: Configure Network Tracing
- Interpreting Network Tracing
- Tracing and Instrumenting Applications

Interpreting Network Tracing

Article • 09/15/2021

When network tracing is enabled, you can use tracing to capture calls your application makes to various System.Net class members. The output from these calls may be similar to the following examples.

Output

```
[588]   (4357)   Entering Socket#33574638::Send()
[588]   (4387)   Exiting Socket#33574638::Send()-> 61#61
```

In the preceding example, [588] is the current thread's unique identifier. (4357) and (4387) are timestamps denoting the number of milliseconds that have elapsed since the application started. The data following the timestamp shows the application entering and exiting the method **Socket.Send**. The object executing the **Send** method has 33574638 as its unique identifier. The method exit trace includes the return value (61 in the preceding example).

Network traces can capture network traffic that is sent from or received by your application using application-level protocols such as Hypertext Transfer Protocol (HTTP). This data can be captured as text and, optionally, hexadecimal data. Hexadecimal data is available when you specify **includehex** as the value of the **tracemode** attribute. (For detailed information about this attribute, see How to: Configure Network Tracing.) The following example trace was generated using **includehex**.

```
[1692] (1142) 00000000 : 47 45 54 20 2F 77 70 61-64 2E 64 61 74 20 48 54 : GET /wpad.dat HT
```

```
[1692] (1142) 00000010 : 54 50 2F 31 2E 31 0D 0A-48 6F 73 74 3A 20 69 74 : TP/1.1..Host: it
```

```
[1692] (1142) 00000020 : 67 70 72 6F 78 79 0D 0A-43 6F 6E 6E 65 63 74 69 : gproxy..Connecti
```

```
[1692] (1142) 00000030 : 6F 6E 3A 20 43 6C 6F 73-65 0D 0A 0D 0A : on: Close....
```

To omit hexadecimal data, specify **protocolonly** as the value for the **tracemode** attribute. The following example shows the trace when **protocolonly** is specified.

```
[2444] (594) Data from ConnectStream#33574638::WriteHeaders<<GET /wpad.dat HTTP/1.1
```

```
Host: itgproxy
```

```
Connection: Close
```

See also

- Enabling Network Tracing
- How to: Configure Network Tracing
- Network Tracing in the .NET Framework

Enabling Network Tracing

Article • 09/15/2021

Network tracing provides access to information about method invocations and network traffic generated by a managed application. You must complete the following tasks to enable network tracing in your application:

- Compile your code with tracing enabled. See How to: Compile Conditionally with Trace and Debug for more information about the compiler switches required to enable tracing.
- Specify a destination for tracing output.
- Configure the behavior of network tracing. See How to: Configure Network Tracing for detailed information.

The most common trace destinations, also referred to as trace listeners, are the default listener and the log file.

Tracing uses the default listener if you do not specify a trace listener. You can view messages sent to the default listener by executing your code in a managed code-enabled debugger such as the CLR Debugger shipped with the .NET Framework SDK, or DBwin32.exe shipped with the Windows SDK. Using the CLR Debugger, the trace messages appear in the **Output** window.

If you prefer to use a file to receive traces, you can specify a log file by using configuration settings, as shown in the following example. (For a general discussion of configuration files, see Configuration Files.)

To send traces to a log file, add the following node to the `<system.diagnostics>` node of the appropriate configuration file (application or machine). You can change the name of the file (trace.log) to suit your needs.

XML

```
<system.diagnostics>
  <trace autoflush="true" indentsize="4">
    <listeners>
      <add name="file" type="System.Diagnostics.TextWriterTraceListener"
initializeData="trace.log"/>
    </listeners>
  </trace>
</system.diagnostics>
```

See also

- Interpreting Network Tracing
- Network Tracing in the .NET Framework
- Tracing and Instrumenting Applications

How to: Configure network tracing

Article • 09/15/2021

The application or computer configuration file holds the settings that determine the format and content of network traces. Before performing this procedure, be sure tracing is enabled. For more information, see Enable network tracing.

The computer configuration file, *machine.config*, is stored in the *%windir%\Microsoft.NET\Framework* folder. There is a separate *machine.config* file in the folders under *%windir%\Microsoft.NET\Framework* for each version of the .NET Framework installed on the computer, for example:

- *C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Config\machine.config*
- *C:\WINDOWS\Microsoft.NET\Framework64\v4.0.30319\Config\machine.config*

These settings can also be made in the configuration file for the application, which has precedence over the computer configuration file.

Configure network tracing

To configure network tracing, add the following lines to the appropriate configuration file. The values and options for these settings are described in the tables below.

XML

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="System.Net" tracemode="includehex" maxdatasize="1024">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
      <source name="System.Net.Cache">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
      <source name="System.Net.Http">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
      <source name="System.Net.Sockets">
        <listeners>
          <add name="System.Net"/>
```

```
        </listeners>
      </source>
      <source name="System.Net.WebSockets">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="System.Net" value="Verbose"/>
      <add name="System.Net.Cache" value="Verbose"/>
      <add name="System.Net.Http" value="Verbose"/>
      <add name="System.Net.Sockets" value="Verbose"/>
      <add name="System.Net.WebSockets" value="Verbose"/>
    </switches>
    <sharedListeners>
      <add name="System.Net"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="network.log"
        traceOutputOptions="ProcessId, DateTime"
      />
    </sharedListeners>
    <trace autoflush="true"/>
  </system.diagnostics>
</configuration>
```

Trace output from methods

When you add a name to the `<switches>` block, the trace output includes information from some methods related to the name. The following table describes the output:

Expand table

Name	Output from
`System.Net.Sockets`	Some public methods of the Socket, TcpListener, TcpClient, and Dns classes.
`System.Net`	Some public methods of the HttpWebRequest, HttpWebResponse, FtpWebRequest, and FtpWebResponse classes, and SSL debug information (invalid certificates, missing issuers list, and client certificate errors).
`System.Net.HttpListener`	Some public methods of the HttpListener, HttpListenerRequest, and HttpListenerResponse classes.
`System.Net.Cache`	Some private and internal methods in `System.Net.Cache`.
`System.Net.Http`	Some public methods of the HttpClient, DelegatingHandler, HttpClientHandler, HttpMessageHandler,

Name	Output from
	MessageProcessingHandler, and WebRequestHandler classes.
`System.Net.WebSockets.WebSocket`	Some public methods of the ClientWebSocket and WebSocket classes.

Trace output attributes

The attributes listed in the following table configure trace output:

Expand table

Attribute name	Attribute value
`value`	Required String attribute. Sets the verbosity of the output. Legitimate values are `Critical`, `Error`, `Verbose`, `Warning`, and `Information`. This attribute must be set on the **add** element of the **switches** element. An exception is thrown if this attribute is set on the **source** element. Example: `<add name="System.Net" value="Verbose"/>`
`maxdatasize`	Optional Int32 attribute. Sets the maximum number of bytes of network data included in each line trace. The default value is 1024. This attribute must be set on the **source** element. An exception is thrown if this attribute is set on an element under the **switches** element. Example: `<source name="System.Net" tracemode="includehex" maxdatasize="1024">`
`tracemode`	Optional String attribute. Set to `includehex` to show protocol traces in hexadecimal and text format. Set to `protocolonly` to show only text. The default value is `includehex`. This attribute must be set on the **source** element. An exception is thrown if this attribute is set on an element under the **switches** element. Example: `<source name="System.Net" tracemode="includehex" maxdatasize="1024">`

See also

- Interpreting Network Tracing

- Network Tracing in the .NET Framework
- Enabling Network Tracing
- Tracing and Instrumenting Applications

Cache Management for Network Applications

Article • 09/15/2021

This topic and its related subtopics describe caching for resources obtained using the WebClient, WebRequest, HttpWebRequest, and FtpWebRequest classes.

A cache provides temporary storage of resources that have been requested by an application. If an application requests the same resource more than once, the resource can be returned from the cache, avoiding the overhead of re-requesting it from the server. Caching can improve application performance by reducing the time required to get a requested resource. Caching can also decrease network traffic by reducing the number of trips to the server. While caching improves performance, it increases the risk that the resource returned to the application is stale, meaning that it is not identical to the resource that would have been sent by the server if caching were not in use.

Caching may allow unauthorized users or processes to read sensitive data. An authenticated response that is cached may be retrieved from the cache without an additional authorization. If caching is enabled, change to CachePolicy to BypassCache or NoCacheNoStore to disable caching for this request.

Due to security concerns, caching is **not** recommended for middle tier scenarios.

In This Section

Cache Policy
Explains what a cache policy is and how to define one.

Location-Based Cache Policies
Defines each type of location-based cache policy available for Hypertext Transfer Protocol (http and https) resources.

Time-Based Cache Policies
Describes the criteria that can be used to customize a time-based cache policy.

Configuring Caching in Network Applications
Describes how to programmatically create cache policies and requests that use caching.

Reference

System.Net.Cache

Defines the types and enumerations used to define cache policies for resources obtained using the WebRequest, HttpWebRequest, and FtpWebRequest classes.

Cache Policy

Article • 09/15/2021

A cache policy defines rules that are used to determine whether a request can be satisfied using a cached copy of the requested resource. Applications specify client cache requirements for freshness, but the effective cache policy is determined by the client cache requirements, the server's content expiration requirements, and the server's revalidation requirements. The interaction of client cache policy and server requirements always results in the most conservative cache policy, to help ensure that the freshest content is returned to the client application.

Cache policies are either location-based or time-based. A location-based cache policy defines the freshness of cached entries based on where the requested resource can be taken from. A time-based cache policy defines the freshness of cached entries using the time the resource was retrieved, headers returned with the resource, and the current time. Most applications can use the default time-based cache policy, which implements the caching policy specified in RFC 2616, available at Internet Engineering Task Force (IETF) website.

The classes described in the following table are used to specify cache policies.

Expand table

Class name	Description
HttpRequestCachePolicy	Represents location-based and time-based cache policies for resources requested using HttpWebRequest objects.
RequestCachePolicy	Represents location-based cache policies or the Default time-based cache policy for resources requested using WebRequest objects.
HttpCacheAgeControl	Specifies values used to create time-based HttpRequestCachePolicy objects.
HttpRequestCacheLevel	Specifies values used to create location-based and time-based HttpRequestCachePolicy objects.
RequestCacheLevel	Specifies values used to create location-based or the Default time-based RequestCachePolicy objects.

You can define a cache policy for all requests made by your application or for individual requests. When you specify both an application-level cache policy and a request-level cache policy, the request-level policy is used. You can specify an application-level cache

policy programmatically or by using the application or machine configuration files. For more information, see <requestCaching> Element (Network Settings).

To create a cache policy, you must create a policy object by creating an instance of the RequestCachePolicy or HttpRequestCachePolicy class. To specify the policy on a request, set the request's CachePolicy property to the policy object. When setting an application-level policy programmatically, set the DefaultCachePolicy property to the policy object.

For code examples that demonstrate creating and using cache policies, see Configuring Caching in Network Applications.

See also

- Cache Management for Network Applications
- Location-Based Cache Policies
- Time-Based Cache Policies
- Configuring Caching in Network Applications

Location-Based Cache Policies

Article • 09/15/2021

A location-based cache policy defines the freshness of valid cached entries based on where the requested resource can be taken from. A cached resource is valid if using it does not does not violate server-specified revalidation requirements. A location-based cache policy is created programmatically by using a RequestCachePolicy or HttpRequestCachePolicy class constructor. The type of location-based policy is passed to the constructor using a RequestCacheLevel or HttpRequestCacheLevel enumeration value. For code examples that create location-based cache policies, see How to: Set a Location-Based Cache Policy for an Application. The following sections explain each type of location-based cache policy for Hypertext Transfer Protocol (http and https) resources.

Cache If Available Policy

If a valid requested resource is in the local cache, the cached resource is used; otherwise, the request for the resource is sent to the server. If the requested resource is available in any cache between the client and the server, the request can be satisfied by an intermediate cache.

Cache Only Policy

If a valid requested resource is in the local cache, the cached resource is used. When this cache policy level is specified, a WebException exception is thrown if the item is not in the local cache.

Cache Or Next Cache Only Policy

If a valid requested resource is in the local cache or an intermediate cache on the local area network, the cached resource is used. Otherwise, a WebException exception is thrown. In the HTTP caching protocol, this is achieved using the only-if-cached cache control directive.

No Cache No Store Policy

A requested resource is never used from any cache and is never placed in any cache. If a requested resource is present in the local cache, it is removed. This policy level indicates

to intermediate caches that they should also remove the resource. In the HTTP caching protocol, this is achieved using the no-store cache control directive.

Refresh Policy

A requested resource can be used if it is obtained from the server or found in a cache other than the local cache. Before the request can be satisfied by an intermediate cache, that cache must revalidate its cached entry with the server. In the HTTP caching protocol, this is achieved using the max-age = 0 cache control directive and the no-cache Pragma header.

Reload Policy

Requested resources must be obtained from the server. The response might be saved in the local cache. In the HTTP caching protocol, this is achieved using the no-cache cache control directive and the no-cache Pragma header.

Revalidate Policy

Compares the copy of the resource in the cache with the copy on the server. If the copy on the server is newer, it is used to satisfy the request and replaces the copy in the cache. If the copy in the cache is the same as the server copy, the cached copy is used. In the HTTP caching protocol, this is achieved using a conditional request.

See also

- Cache Management for Network Applications
- Cache Policy
- Time-Based Cache Policies
- Configuring Caching in Network Applications
- <requestCaching> Element (Network Settings)

Time-Based Cache Policies

Article • 09/15/2021

A time-based cache policy defines the freshness of cached entries using the time the resource was retrieved, the headers returned with the resource, and the current time. When setting a time-based cache policy, you can either use the Default time-based policy or create a customized time-based policy. When using the default time-based policy for resources obtained using Hypertext Transfer Protocol (HTTP), the exact cache behavior is determined by the headers included in the cached response and by the behaviors specified in sections 13 and 14 of RFC 2616, available at Internet Engineering Task Force (IETF) website. For a code example that demonstrates setting the default time-based policy for HTTP resources, see How to: Set the Default Time-Based Cache Policy for an Application. For code examples that demonstrate creating and using cache policies, see Configuring Caching in Network Applications.

Criteria to Determine Freshness of Cached Entries

To customize a time-based cache policy, you can specify that one or more of the following criteria be used to determine the freshness of cached entries:

- Maximum age
- Maximum staleness
- Minimum freshness
- Cache synchronization date

> **Note**
>
> Using the default time-based cache policy should not be confused with setting a default cache policy for your application. The default time-based policy is a specific policy that can be used at the request or application level. The default cache policy for your application is a policy (location-based or time-based) that takes effect when no policy is set on a request. For details on setting a default cache policy for your application, see DefaultCachePolicy.

Maximum Age

The maximum age policy criterion specifies the amount of time a cached copy of a resource can be used. If the cached copy of the resource is older than the amount of time specified, the resource must be revalidated by checking it against the content on the server. If the maximum age would allow the resource to be used after it expires, this criteria is not honored unless a maximum staleness value is also specified.

Maximum Staleness

The maximum staleness policy criterion specifies the length of time after content expiration that the cached copy of the resource can be used. This is the only cache policy criterion that permits resources to be used after they have expired.

Minimum Freshness

The minimum freshness policy criterion specifies the length of time before content expiration that the cached copy of the resource can be used. This policy has the effect of causing a cache entry to expire before its expiration date; therefore, the minimum freshness and maximum staleness settings are mutually exclusive.

Cache Synchronization Date

The cache synchronization date policy criterion determines when a cached copy of a resource must be revalidated by checking it against the content on the server. If the content has changed since the item was cached, it is retrieved from the server, stored in the cache, and returned to the application. If the content has not changed, its timestamp is updated and the application gets the cached content.

The cache synchronization date allows you to specify an absolute date when cached contents must be revalidated. If a fresh cache entry was last revalidated prior to the cache synchronization date, revalidation with the server still occurs. If the cache entry was revalidated after the cache synchronization date and there are no additional freshness or server revalidation requirements that invalidate the cached entry, the entry from the cache is used. If the cache synchronization date is set to a future date, the entry is revalidated every time it is requested, until the cache synchronization date passes.

The following topics provide information about the effects of combining time-based cache policy criteria:

- Cache Policy Interaction—Maximum Age and Maximum Staleness
- Cache Policy Interaction—Maximum Age and Minimum Freshness

See also

- Cache Management for Network Applications
- Cache Policy
- Location-Based Cache Policies
- Configuring Caching in Network Applications
- <requestCaching> Element (Network Settings)

Cache Policy Interaction—Maximum Age and Maximum Staleness

Article • 09/15/2021

To help ensure that the freshest content is returned to the client application, the interaction of client cache policy and server revalidation requirements always results in the most conservative cache policy. All the examples in this topic illustrate the cache policy for a resource that is cached on January 1 and expires on January 4.

In the following examples, the maximum staleness value (`maxStale`) is used in conjunction with a maximum age (`maxAge`):

- If the cache policy sets `maxAge` = 5 days and does not specify a `maxStale` value, according to the `maxAge` value, the content is usable until January 6. However, according to the server's revalidation requirements, the content expires on January 4. Because the content expiration date is more conservative (sooner), it takes precedence over the `maxAge` policy. Therefore, the content expires on January 4 and must be revalidated even though its maximum age has not been reached.
- If the cache policy sets `maxAge` = 5 days and `maxStale` = 3 days, according to the `maxAge` value, the content is usable until January 6. According to the `maxStale` value, the content is usable until January 7. Therefore, the content gets revalidated on January 6.
- If the cache policy sets `maxAge` = 5 days and `maxStale` = 1 day, according to the `maxAge` value, the content is usable until January 6. According to the `maxStale` value, the content is usable until January 5. Therefore, the content gets revalidated on January 5.

When the maximum age is less than the content expiration date, the more conservative caching behavior always prevails and the maximum staleness value has no effect. The following examples illustrate the effect of setting a maximum staleness (`maxStale`) value when the maximum age (`maxAge`) is reached before the content expires:

- If the cache policy sets `maxAge` = 1 day and does not specify a value for `maxStale` value, the content is revalidated on January 2 even though it has not expired.
- If the cache policy sets `maxAge` = 1 day and `maxStale` = 3 days, the content is revalidated on January 2 to enforce the more conservative policy setting.

- If the cache policy sets `maxAge` = 1 day and `maxStale` = 1 day, the content is revalidated on January 2.

See also

- Cache Management for Network Applications
- Cache Policy
- Location-Based Cache Policies
- Time-Based Cache Policies
- Configuring Caching in Network Applications
- Cache Policy Interaction—Maximum Age and Minimum Freshness

Cache Policy Interaction—Maximum Age and Minimum Freshness

Article • 09/15/2021

To help ensure that the freshest content is returned to the client application, the interaction of client cache policy and server revalidation requirements always results in the most conservative cache policy. All the examples in this topic illustrate the cache policy for a resource that is cached on January 1 and expires on January 4.

The following examples illustrate the cache policy that results from the interaction of the maximum age (`maxAge`) and minimum freshness (`minFresh`) values.

- If the cache policy sets `maxAge` = 2 days and `minFresh` is not specified, the content is revalidated on January 3.
- If the cache policy sets `maxAge` = 2 days and `minFresh` = 1 day, according to `maxAge`, the content is fresh until January 3. According to `minFresh`, the content is fresh until January 3. Therefore, the content must be revalidated on January 3.
- If the cache policy sets `maxAge` = 2 days and `minFresh` = 2 days, according to `maxAge`, the content is fresh until January 3. According to `minFresh` the content is fresh until January 2. Therefore, the content must be revalidated on January 2.

See also

- Cache Management for Network Applications
- Cache Policy
- Location-Based Cache Policies
- Time-Based Cache Policies
- Configuring Caching in Network Applications
- Cache Policy Interaction—Maximum Age and Maximum Staleness

Configuring Caching in Network Applications

Article • 09/15/2021

To configure caching, you must specify a cache policy at the application or WebRequest level. The following topics provide code examples that demonstrate configuring applications and requests to use caching.

- How to: Set a Location-Based Cache Policy for an Application
- How to: Set the Default Time-Based Cache Policy for an Application
- How to: Customize a Time-Based Cache Policy
- How to: Set Cache Policy for a Request

You can also configure cache policy using application or machine configuration files. For more information, see | <requestCaching> Element (Network Settings).

See also

- Cache Management for Network Applications
- Cache Policy
- Location-Based Cache Policies
- Time-Based Cache Policies

How to: Set a Location-Based Cache Policy for an Application

Article • 09/15/2021

Location-based cache policies allow an application to explicitly define caching behavior based on the location of the requested resource. This topic demonstrates setting the cache policy programmatically. For information on setting the policy for an application using the configuration files, see <requestCaching> Element (Network Settings).

To set a location-based cache policy for an application

1. Create a RequestCachePolicy or HttpRequestCachePolicy object.
2. Set the policy object as the default for the application domain.

To set a policy that takes requested resources from a cache

- Create a policy that takes requested resources from a cache if available, and otherwise, sends requests to the server, by setting the cache level to CacheIfAvailable. A request can be fulfilled by any cache between the client and server, including remote caches.

```
C#
```

```
public static void UseCacheIfAvailable()
{
    HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.CacheIfAvailable);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

To set a policy that prevents any cache from supplying resources

- Create a policy that prevents any cache from supplying requested resources by setting the cache level to NoCacheNoStore. This policy level removes the resource from the local cache if it is present and indicates to remote caches that they should also remove the resource.

```
C#

public static void DoNotUseCache()
{
HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.NoCacheNoStore);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

To set a policy that returns requested resources only if they are in the local cache

- Create a policy that returns requested resources only if they are in the local cache by setting the cache level to CacheOnly. If the requested resource is not in the cache, a WebException exception is thrown.

```
C#

public static void OnlyUseCache()
{
    HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.CacheOnly);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

To set a policy that prevents the local cache from supplying resources

- Create a policy that prevents the local cache from supplying requested resources by setting the cache level to Refresh. If the requested resource is in an intermediate cache and is successfully revalidated, the intermediate cache can supply the requested resource.

```
C#

public static void DoNotUseLocalCache()
{
 HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.Refresh);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

To set a policy that prevents any cache from supplying requested resources

- Create a policy that prevents any cache from supplying requested resources by setting the cache level to Reload. The resource returned by the server can be stored in the cache.

```
C#

public static void SendToServer()
{
HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.Reload);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

To set a policy that allows any cache to supply requested resources if the resource on the server is not newer than the cached copy

- Create a policy that allows any cache to supply requested resources if the resource on the server is not newer than the cached copy by setting the cache level to Revalidate.

```
C#

public static void CheckServer()
{
HttpRequestCachePolicy policy = new HttpRequestCachePolicy
         (HttpRequestCacheLevel.Revalidate);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

See also

- Cache Management for Network Applications
- Cache Policy
- Location-Based Cache Policies
- Time-Based Cache Policies
- <requestCaching> Element (Network Settings)

How to: Set the Default Time-Based Cache Policy for an Application

Article • 09/15/2021

The default time-based cache policy allows an application to have its cache behavior defined by the headers sent with the cached resource and the cache behavior defined in sections 13 and 14 of RFC 2616, available at Internet Engineering Task Force (IETF) website. This is the appropriate cache behavior for most applications.

To set the default automatic policy for an application

1. Create a default time-based policy object.

2. Set the policy object as the default for the application domain.

Example

The two examples in this section produce identical policies.

The following example creates a default time-based policy and sets it as the default for the application domain.

C#

```
public static void SetDefaultTimeBasedPolicy ()
{
    HttpRequestCachePolicy policy = new HttpRequestCachePolicy ();
    HttpWebRequest.DefaultCachePolicy = policy ;
}
```

You can also create the default time-based cache policy using the RequestCachePolicy class as shown in the following example:

C#

```
public static void SetDefaultTimeBasedPolicy2()
{
    RequestCachePolicy policy = new RequestCachePolicy ();
    HttpWebRequest.DefaultCachePolicy = policy ;
}
```

See also

- Cache Management for Network Applications
- Cache Policy
- Location-Based Cache Policies
- Time-Based Cache Policies
- <requestCaching> Element (Network Settings)

How to: customize a time-based cache policy

Article • 09/15/2021

When creating a time-based cache policy, you can customize caching behavior by specifying values for maximum age, minimum freshness, maximum staleness, or cache synchronization date. The HttpRequestCachePolicy object provides several constructors that allow you to specify valid combinations of these values.

To create a time-based cache policy that uses a cache synchronization date

Create a time-based cache policy that uses a cache synchronization date by passing a DateTime object to the HttpRequestCachePolicy constructor:

C#

```
public static HttpRequestCachePolicy CreateLastSyncPolicy(DateTime when)
{
    var policy = new HttpRequestCachePolicy(when);
    Console.WriteLine("When: {0}", when);
    Console.WriteLine(policy.ToString());
    return policy;
}
```

The output is similar to the following:

Output

```
When: 1/14/2004 8:07:30 AM
Level:Default CacheSyncDate:1/14/2004 8:07:30 AM
```

To create a time-based cache policy that is based on minimum freshness

Create a time-based cache policy that is based on minimum freshness by specifying MinFresh as the `cacheAgeControl` parameter value and passing a TimeSpan object to the HttpRequestCachePolicy constructor:

C#

```
public static HttpRequestCachePolicy CreateMinFreshPolicy(TimeSpan span)
{
    var policy = new HttpRequestCachePolicy(HttpCacheAgeControl.MinFresh,
span);
    Console.WriteLine(policy.ToString());
    return policy;
}
```

For the following invocation:

C#

```
CreateMinFreshPolicy(new TimeSpan(1,0,0));
```

The output is:

Output

```
Level:Default MinFresh:3600
```

To create a time-based cache policy that is based on minimum freshness and maximum age

Create a time-based cache policy that is based on minimum freshness and maximum age by specifying MaxAgeAndMinFresh as the `cacheAgeControl` parameter value and passing two TimeSpan objects to the HttpRequestCachePolicy constructor, one to specify the maximum age for resources and a second to specify the minimum freshness permitted for an object returned from the cache:

C#

```
public static HttpRequestCachePolicy CreateFreshAndAgePolicy(TimeSpan
freshMinimum, TimeSpan ageMaximum)
{
    var policy = new
HttpRequestCachePolicy(HttpCacheAgeControl.MaxAgeAndMinFresh, ageMaximum,
freshMinimum);
    Console.WriteLine(policy.ToString());
    return policy;
}
```

For the following invocation:

C#

```
CreateFreshAndAgePolicy(new TimeSpan(5,0,0), new TimeSpan(10,0,0));
```

The output is:

Output

```
Level:Default MaxAge:36000 MinFresh:18000
```

See also

- Cache Management for Network Applications
- Cache Policy
- Location-Based Cache Policies
- Time-Based Cache Policies
- <requestCaching> Element (Network Settings)

How to: Set Cache Policy for a Request

Article • 09/15/2021

The following example demonstrates setting a cache policy for a request. The example input is a URI such as `http://www.contoso.com/`.

Example

The following code example creates a cache policy that allows the requested resource to be used from the cache if it has not been in the cache for longer than one day. The example displays a message that indicates whether the resource was used from the cache—for example, `"The response was retrieved from the cache : False."`—and then displays the resource. A request can be fulfilled by any cache between the client and server.

C#

```
using System;
using System.Net;
using System.Net.Cache;
using System.IO;

namespace Examples.System.Net.Cache
{
    public class CacheExample
    {
        public static void UseCacheForOneDay(Uri resource)
        {
            // Create a policy that allows items in the cache
            // to be used if they have been cached one day or less.
            HttpRequestCachePolicy requestPolicy =
                new HttpRequestCachePolicy (HttpCacheAgeControl.MaxAge,
                TimeSpan.FromDays(1));

            WebRequest request = WebRequest.Create (resource);
            // Set the policy for this request only.
            request.CachePolicy = requestPolicy;
            HttpWebResponse response =
(HttpWebResponse)request.GetResponse();
            // Determine whether the response was retrieved from the cache.
            Console.WriteLine ("The response was retrieved from the cache :
{0}.",
               response.IsFromCache);
            Stream s = response.GetResponseStream ();
            StreamReader reader = new StreamReader (s);
            // Display the requested resource.
            Console.WriteLine(reader.ReadToEnd());
            reader.Close ();
```

```
            s.Close();
            response.Close();
        }
        public static void Main(string[] args)
        {
            if (args == null || args.Length != 1)
            {
                Console.WriteLine ("You must specify the URI to retrieve.");
                return;
            }
            UseCacheForOneDay (new Uri(args[0]));
        }
    }
}
```

See also

- Cache Management for Network Applications
- Cache Policy
- Location-Based Cache Policies
- Time-Based Cache Policies
- <requestCaching> Element (Network Settings)

FTP

Article • 08/27/2022

.NET Framework provides comprehensive support for the FTP protocol with the FtpWebRequest and FtpWebResponse classes. These classes are derived from WebRequest and WebResponse. In most cases, the WebRequest and WebResponse classes provide all that's necessary to make the request, but if you need access to the FTP-specific features exposed as properties, you can typecast these classes to FtpWebRequest or FtpWebResponse.

> **Note**
>
> This article is specific to projects that target .NET Framework. For projects that target .NET 6 and later versions, **FTP is no longer supported**.

Examples

For more information, see the following topics: How to: Download Files with FTP, How to: Upload Files with FTP, and How to: List Directory Contents with FTP.

FTP and proxies

If a proxy (specified by the Proxy property) is an HTTP proxy, then only the DownloadFile, ListDirectory, and ListDirectoryDetails commands are supported.

FTP

Article • 08/27/2022

.NET Framework provides comprehensive support for the FTP protocol with the FtpWebRequest and FtpWebResponse classes. These classes are derived from WebRequest and WebResponse. In most cases, the WebRequest and WebResponse classes provide all that's necessary to make the request, but if you need access to the FTP-specific features exposed as properties, you can typecast these classes to FtpWebRequest or FtpWebResponse.

> **Note**
>
> This article is specific to projects that target .NET Framework. For projects that target .NET 6 and later versions, **FTP is no longer supported**.

Examples

For more information, see the following topics: How to: Download Files with FTP, How to: Upload Files with FTP, and How to: List Directory Contents with FTP.

FTP and proxies

If a proxy (specified by the Proxy property) is an HTTP proxy, then only the DownloadFile, ListDirectory, and ListDirectoryDetails commands are supported.

How to: Download files with FTP

Article • 04/20/2022

This sample shows how to download a file from an FTP server.

> **Note**
>
> This article is specific to projects that target .NET Framework. For projects that target .NET 6 and later versions, FTP is no longer supported.

Example

C#

```
using System;
using System.IO;
using System.Net;

namespace Examples.System.Net
{
    public class WebRequestGetExample
    {
        public static void Main ()
        {
            // Get the object used to communicate with the server.
            FtpWebRequest request =
(FtpWebRequest)WebRequest.Create("ftp://www.contoso.com/test.htm");
            request.Method = WebRequestMethods.Ftp.DownloadFile;

            // This example assumes the FTP site uses anonymous logon.
            request.Credentials = new
NetworkCredential("anonymous","janeDoe@contoso.com");

            FtpWebResponse response = (FtpWebResponse)request.GetResponse();

            Stream responseStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(responseStream);
            Console.WriteLine(reader.ReadToEnd());

            Console.WriteLine($"Download Complete, status
{response.StatusDescription}");

            reader.Close();
            response.Close();
        }
    }
}
```

How to: Upload files with FTP

Article • 05/15/2022

This sample shows how to upload a file to an FTP server.

> **Note**
>
> This article is specific to projects that target .NET Framework. For projects that target .NET 6 and later versions, FTP is no longer supported.

Example

C#

```
using System;
using System.IO;
using System.Net;
using System.Threading.Tasks;

namespace Examples.System.Net
{
    public class WebRequestGetExample
    {
        public static async Task Main()
        {
            // Get the object used to communicate with the server.
            FtpWebRequest request =
(FtpWebRequest)WebRequest.Create("ftp://www.contoso.com/test.htm");
            request.Method = WebRequestMethods.Ftp.UploadFile;

            // This example assumes the FTP site uses anonymous logon.
            request.Credentials = new NetworkCredential("anonymous",
"janeDoe@contoso.com");

            // Copy the contents of the file to the request stream.
            using (FileStream fileStream = File.Open("testfile.txt",
FileMode.Open, FileAccess.Read))
            {
                using (Stream requestStream = request.GetRequestStream())
                {
                    await fileStream.CopyToAsync(requestStream);
                    using (FtpWebResponse response =
(FtpWebResponse)request.GetResponse())
                    {
                        Console.WriteLine($"Upload File Complete, status
{response.StatusDescription}");
                    }
```

```
                }
            }
        }
    }
}
```

How to: List directory contents with FTP

Article • 04/20/2022

This sample shows how to list the directory contents of an FTP server.

> **Note**
>
> This article is specific to projects that target .NET Framework. For projects that target .NET 6 and later versions, FTP is no longer supported.

Example

C#

```
using System;
using System.IO;
using System.Net;

namespace Examples.System.Net
{
    public class WebRequestGetExample
    {
        public static void Main ()
        {
            // Get the object used to communicate with the server.
            FtpWebRequest request =
(FtpWebRequest)WebRequest.Create("ftp://www.contoso.com/");
            request.Method = WebRequestMethods.Ftp.ListDirectoryDetails;

            // This example assumes the FTP site uses anonymous logon.
            request.Credentials = new NetworkCredential
("anonymous","janeDoe@contoso.com");

            FtpWebResponse response = (FtpWebResponse)request.GetResponse();

            Stream responseStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(responseStream);
            Console.WriteLine(reader.ReadToEnd());

            Console.WriteLine($"Directory List Complete, status
{response.StatusDescription}");

            reader.Close();
            response.Close();
        }
    }
}
```

If you need to list a specific directory, just add the directory to the end of the URI you're using in the WebRequest.Create method:

```
C#

FtpWebRequest request =
(FtpWebRequest)WebRequest.Create("ftp://www.contoso.com/your_preferred_direc
tory");
```

System.Net.FtpWebRequest.Proxy property

Article • 05/10/2024

This article provides supplementary remarks to the reference documentation for this API.

> **Note**
>
> This property is not supported on .NET Core, and setting it has no effect. Getting the property value returns `null`.

The Proxy property identifies the IWebProxy instance that communicates with the FTP server. The proxy is set by the system by using configuration files and the Local Area Network settings. To specify that no proxy should be used, set Proxy to the proxy instance returned by the GlobalProxySelection.GetEmptyWebProxy method. For more information about automatic proxy detection, see Automatic Proxy Detection.

You must set Proxy before writing data to the request's stream or getting the response. Changing Proxy after calling the GetRequestStream, BeginGetRequestStream, GetResponse, or BeginGetResponse method causes an InvalidOperationException exception.

The FtpWebRequest class supports HTTP and ISA Firewall Client proxies.

If the specified proxy is an HTTP proxy, only the DownloadFile, ListDirectory, and ListDirectoryDetails commands are supported.

Use UDP services

Article • 09/15/2021

The UdpClient class communicates with network services using UDP. The properties and methods of the UdpClient class abstract the details of creating a Socket for requesting and receiving data using UDP.

User Datagram Protocol (UDP) is a simple protocol that makes a best effort to deliver data to a remote host. However, because the UDP protocol is a connectionless protocol, UDP datagrams sent to the remote endpoint are not guaranteed to arrive, nor are they guaranteed to arrive in the same sequence in which they are sent. Applications that use UDP must be prepared to handle missing, duplicate, and out-of-sequence datagrams.

To send a datagram using UDP, you must know the network address of the network device hosting the service you need and the UDP port number that the service uses to communicate. The Internet Assigned Numbers Authority (IANA) defines port numbers for common services (see Service Name and Transport Protocol Port Number Registry). Services not on the IANA list can have port numbers in the range 1,024 to 65,535.

Special network addresses are used to support UDP broadcast messages on IP-based networks. The following discussion uses the IP version 4 address family used on the Internet as an example.

IP version 4 addresses use 32 bits to specify a network address. For class C addresses using a netmask of 255.255.255.0, these bits are separated into four octets. When expressed in decimal, the four octets form the familiar dotted-quad notation, such as 192.168.100.2. The first two octets (192.168 in this example) form the network number, the third octet (100) defines the subnet, and the final octet (2) is the host identifier.

Setting all the bits of an IP address to one, or 255.255.255.255, forms the limited broadcast address. Sending a UDP datagram to this address delivers the message to any host on the local network segment. Because routers never forward messages sent to this address, only hosts on the network segment receive the broadcast message.

Broadcasts can be directed to specific portions of a network by setting all bits of the host identifier. For example, to send a broadcast to all hosts on the network identified by IP addresses starting with 192.168.1, use the address 192.168.1.255.

The following code example uses a UdpClient to listen for UDP datagrams on port 11,000. The client receives a message string and writes the message to the console.

C#

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

public class UDPListener
{
    private const int listenPort = 11000;

    private static void StartListener()
    {
        UdpClient listener = new UdpClient(listenPort);
        IPEndPoint groupEP = new IPEndPoint(IPAddress.Any, listenPort);

        try
        {
            while (true)
            {
                Console.WriteLine("Waiting for broadcast");
                byte[] bytes = listener.Receive(ref groupEP);

                Console.WriteLine($"Received broadcast from {groupEP} :");
                Console.WriteLine($" {Encoding.ASCII.GetString(bytes, 0,
bytes.Length)}");
            }
        }
        catch (SocketException e)
        {
            Console.WriteLine(e);
        }
        finally
        {
            listener.Close();
        }
    }

    public static void Main()
    {
        StartListener();
    }
}
```

The following code example uses a Socket to send UDP datagrams to the directed broadcast address 192.168.1.255, using port 11,000. The client sends the message string specified on the command line.

C#

```
using System;
using System.Net;
```

```
using System.Net.Sockets;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);

        IPAddress broadcast = IPAddress.Parse("192.168.1.255");

        byte[] sendbuf = Encoding.ASCII.GetBytes(args[0]);
        IPEndPoint ep = new IPEndPoint(broadcast, 11000);

        s.SendTo(sendbuf, ep);

        Console.WriteLine("Message sent to the broadcast address");
    }
}
```

See also

- UdpClient
- IPAddress

Security in network programming

Article • 08/27/2022

The .NET Framework System.Net namespace classes provide built-in support for popular Internet application authentication mechanisms and for .NET Framework code access permissions.

In this section

Transport Layer Security (TLS) best practices with .NET Framework
Describes TLS best practices with the .NET Framework.

Using Secure Sockets Layer
Describes how to use Secure Sockets Layer (SSL) connections.

Internet Authentication
Describes how to use HTTP authentication methods to establish authenticated connections to HTTP servers.

Web and Socket Permissions
Describes how to set code access security for applications that use Internet connections.

Related sections

Network programming in .NET
Introduces the classes in the System.Net and System.Net.Sockets namespaces.

Transport Layer Security (TLS) best practices with .NET Framework

> **Note**
>
> This page contains .NET Framework TLS information. If you're looking for .NET TLS information, see: TLS/SSL Best Practices

.NET Framework supports the use of the Transport Layer Security (TLS) protocol to secure network communications.

What is Transport Layer Security (TLS)?

> **Warning**
>
> TLS 1.0 and 1.1 has been deprecated by RFC8996. This document covers TLS 1.2 and TLS 1.3 only.

The Transport Layer Security (TLS) protocol is an industry latest version of the standard designed to help protect the privacy of information communicated over the Internet. TLS 1.3 is a standard that provides security improvements over previous versions. This article presents recommendations to secure .NET Framework applications that use the TLS protocol.

Who can benefit from this document?

- Directly using the System.Net APIs (for example, System.Net.Http.HttpClient and System.Net.Security.SslStream).
- Directly using WCF clients and services using the System.ServiceModel namespace.

TLS support in .NET Framework

Since the .NET Framework is dependent on `Schannel` on Windows, which versions can be negotiated and which version will be used depends on the operating system.

Here is an updated example table showing the highest supported TLS version for different combinations of operating system versions and .NET Framework target versions:

Expand table

.NET Framework Target Version	Windows 10	Windows 11
3.5	TLS 1.2	TLS 1.2
4.6.2	TLS 1.2	**TLS 1.3**
4.7	TLS 1.2	**TLS 1.3**
4.7.1	TLS 1.2	**TLS 1.3**
4.7.2	TLS 1.2	**TLS 1.3**
4.8	TLS 1.2	**TLS 1.3**
4.8.1	TLS 1.2	**TLS 1.3**

For more information see TLS protocol version support in Schannel.

Recommendations

- For TLS 1.3, target .NET Framework 4.8 or later. Check Audit your code section how to verify your `target framework`.
- Do not specify the TLS version explicitly, i.e. don't use the method overloads of `SslStream` that take an explicit `SslProtocols` parameter.
  - That way your code will let the OS decide on the TLS version.
  - If you must set ServicePointManager.SecurityProtocol, then set it to SecurityProtocolType.SystemDefault. That will also use OS default.
  - If you must use the method overloads of `SslStream` that take an explicit `SslProtocols` parameter, then pass `SslProtocols.SystemDefault` as argument. That will also use OS default.
- Perform a thorough code audit to verify you're not specifying a TLS or SSL version explicitly.

> ⚠ **Warning**
>
> Do not use `SslProtocols.Default`, because it sets TLS version to SSL3 and TLS 1.0 which are obsoleted.

When your app lets the OS choose the TLS version:

- It automatically takes advantage of new TLS protocols added in the future.
- The OS blocks protocols that are discovered not to be secure (for example, SSL3 and TLS 1.0).

This article explains how to enable the strongest security available for the version of .NET Framework that your app targets and runs on. When an app explicitly sets a security protocol and version, it opts out of any other alternative, and opts out of .NET Framework and OS default behavior. If you want your app to be able to negotiate a TLS 1.3 connection, explicitly setting to a lower TLS version prevents a TLS 1.3 connection.

If you can't avoid specifying a protocol version explicitly, we strongly recommend that you specify TLS 1.2 or TLS 1.3 (which is `currently considered secure`). For guidance on identifying and removing TLS 1.0 dependencies, download the Solving the TLS 1.0 Problem white paper.

WCF supports TLS 1.2 as the default in .NET Framework 4.7. Starting with .NET Framework 4.7.1, WCF defaults to the operating system configured version. If an application is explicitly configured with `SslProtocols.None`, WCF uses the operating system default setting when using the NetTcp transport.

You can ask questions about this document in the GitHub issue Transport Layer Security (TLS) best practices with the .NET Framework.

Audit your code and make code changes

For ASP.NET applications, inspect the `<system.web><httpRuntime targetFramework>` element of *web.config* to verify you're using the targeting intended version of the .NET Framework.

For Windows Forms and other applications, see How to: Target a Version of the .NET Framework.

Use the following sections to verify you're not using a specific TLS or SSL version.

Explicitly set a security protocol

If you must explicitly set a security protocol instead of letting .NET or the OS pick the security protocol, pick these protocols:

- For .NET Framework 3.5: TLS 1.2
- For .NET Framework 4.6.2 or later: TLS 1.3

If you can't find specified protocols in enum, you can add those as an extension file. See below:

`SslProtocolExtensions.cs`

c#

```
namespace System.Security.Authentication
{
    public static class SslProtocolsExtensions
    {
        // For .NET Framework 3.5
        public const SslProtocols Tls12 = (SslProtocols)3072;
        // For .NET Framework 4.6.2 and later
        public const SslProtocols Tls13 = (SslProtocols)12288;
    }
}
```

`SecurityProtocolExtensions.cs`

c#

```
using System.Security.Authentication;

namespace System.Net
{
    public static class SecurityProtocolTypeExtensions
    {
        // For .NET Framework 3.5
        public const SecurityProtocolType Tls12 =
(SecurityProtocolType)SslProtocolsExtensions.Tls12;
        // For .NET Framework 4.6.2 and later
        public const SecurityProtocolType Tls13 =
(SecurityProtocolType)SslProtocolsExtensions.Tls13;
    }
}
```

For more information, see Support for TLS System Default Versions included in .NET Framework 3.5 on Windows 8.1 and Windows Server 2012 R2 .

For System.Net APIs (HttpClient, SslStream)

The following sections show how to configure your application to use "currently considered secure versions" of TLS (namely TLS 1.2 and TLS 1.3) if it targets .NET Framework 4.7 or later.

For HttpClient and HttpWebRequest

ServicePointManager uses the default security protocol configured in the OS. To get the default OS choice, if possible, don't set a value for the ServicePointManager.SecurityProtocol property, which defaults to SecurityProtocolType.SystemDefault.

Because the SecurityProtocolType.SystemDefault setting causes the ServicePointManager to use the default security protocol configured by the operating system, your application might run differently based on the OS it's run on. For example, Windows 10 uses TLS 1.2, while Windows 11 uses TLS 1.3.

For SslStream

SslStream defaults to the security protocol and version chosen by the OS. To get the default OS best choice, if possible, don't use the method overloads of SslStream that take an explicit SslProtocols parameter. Otherwise, pass SslProtocols.None. We recommend that you don't use Default; setting `SslProtocols.Default` forces the use of SSL 3.0 /TLS 1.0 and prevents TLS 1.2.

Don't set a value for the SecurityProtocol property (for HTTP networking).

Don't use the method overloads of SslStream that take an explicit SslProtocols parameter (for TCP sockets networking). When you retarget your app to .NET Framework 4.7 or later versions, you'll be following the best practices recommendation.

For WCF apps

The following sections show how to configure your application to use "currently considered secure versions" of TLS (namely, TLS 1.2 and TLS 1.3).

.NET Framework 4.7 and later

Using TCP transport using transport security with certificate credentials

WCF uses the same networking stack as the rest of .NET Framework.

If you're targeting 4.7.1, WCF is configured to allow the OS to choose the best security protocol by default unless explicitly configured:

- In your application configuration file.
- **Or**, in your application in the source code.

By default, .NET Framework 4.7 and later versions are configured to use TLS 1.2 and allow connections using TLS 1.1 or TLS 1.0. Configure WCF to allow the OS to choose the best security protocol by configuring your binding to use SslProtocols.None. You can set this on SslProtocols. `SslProtocols.None` can be accessed from Transport. `NetTcpSecurity.Transport` can be accessed from Security.

If you're using a custom binding:

- Configure WCF to allow the OS to choose the best security protocol by setting SslProtocols to use SslProtocols.None.
- **Or** configure the protocol used with the configuration path `system.serviceModel/bindings/customBinding/binding/sslStreamSecurity:sslProtocols`.

If you're **not** using a custom binding **and** you're setting your WCF binding using configuration, set the protocol used with the configuration path `system.serviceModel/bindings/netTcpBinding/binding/security/transport:sslProtocols`.

Using Message Security with certificate credentials

.NET Framework 4.7 and later versions by default use the protocol specified in the SecurityProtocol property. When the AppContextSwitch `Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols` is set to `true`, WCF chooses the best protocol, up to TLS 1.0.

Configure security via AppContext switches

The AppContext switches described in this section are relevant if your app targets, or runs on, .NET Framework 4.6.2 or a later version. Whether by default, or by setting them explicitly, the switches should be `false` if possible. If you want to configure security via one or both switches, then don't specify a security protocol value in your code; doing so overrides the switches.

HttpClient and SslStream

The switches have the same effect whether you're doing HTTP networking (ServicePointManager) or TCP sockets networking (SslStream).

Switch.System.Net.DontEnableSchUseStrongCrypto

A value of `false` for `Switch.System.Net.DontEnableSchUseStrongCrypto` causes your app to use strong cryptography. A value of `false` for `DontEnableSchUseStrongCrypto` uses more secure network protocols (TLS 1.2 and TLS 1.1) and blocks protocols that are not secure. For more info, see The SCH_USE_STRONG_CRYPTO flag. A value of `true` disables strong cryptography for your app. This switch affects only client (outgoing) connections in your application.

If your app targets .NET Framework 4.6.2 or later versions, this switch defaults to `false`. That's a secure default, which we recommend. If your app runs on .NET Framework 4.6.2, but targets an earlier version, the switch defaults to `true`. In that case, you should explicitly set it to `false`.

`DontEnableSchUseStrongCrypto` should only have a value of `true` if you need to connect to legacy services that don't support strong cryptography and can't be upgraded.

Switch.System.Net.DontEnableSystemDefaultTlsVersions

A value of `false` for `Switch.System.Net.DontEnableSystemDefaultTlsVersions` causes your app to allow the operating system to choose the protocol. A value of `true` causes your app to use protocols picked by the .NET Framework.

If your app targets .NET Framework 4.7 or later versions, this switch defaults to `false`. That's a secure default that we recommend. If your app runs on .NET Framework 4.7 or later versions, but targets an earlier version, the switch defaults to `true`. In that case, you should explicitly set it to `false`.

Configure Schannel protocols in the Windows Registry

You can use the registry for fine-grained control over the protocols that your client or server app negotiates. Your app's networking goes through Schannel (which is another name for Secure Channel). By configuring `Schannel`, you can configure your app's behavior.

Start with the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Protocols` registry key. Under that key you can create any subkeys in the set `TLS 1.2`, `TLS 1.3`. Under each of those subkeys, you can create subkeys `Client` and/or `Server`. Under `Client` and `Server`, you can create DWORD values `DisabledByDefault` (0 or 1) and `Enabled` (0 or 1).

For more information see: TLS Registry Settings - Schannel

The SCH_USE_STRONG_CRYPTO flag

When it's enabled (by default, or by an AppContext switch), .NET Framework uses the `SCH_USE_STRONG_CRYPTO` flag when your app initiates a TLS connection to a server. .NET Framework passes the flag to `Schannel` to instruct it to disable known weak cryptographic algorithms, cipher suites, and TLS/SSL protocol versions that may be otherwise enabled for better interoperability. For more information, see:

- Secure Channel
- SCHANNEL_CRED structure

The `SCH_USE_STRONG_CRYPTO` flag is also passed to `Schannel` for client (outgoing) connections when you explicitly use the `Tls11` or `Tls12` enumerated values of SecurityProtocolType or SslProtocols. The `SCH_USE_STRONG_CRYPTO` flag is used only for connections where your application acts the role of the client. You can disable weak protocols and algorithms when your applications acts the role of the server by configuring the machine-wide `Schannel` registry settings.

Last updated on 04/02/2026

Using Secure Sockets Layer

Article • 08/27/2022

The System.Net classes use the Secure Sockets Layer (SSL) to encrypt the connection for several network protocols.

For http connections, the WebRequest and WebResponse classes use SSL to communicate with web hosts that support SSL. The decision to use SSL is made by the WebRequest class, based on the URI it is given. If the URI begins with "https:", SSL is used; if the URI begins with "http:", an unencrypted connection is used.

To use SSL with File Transfer Protocol (FTP), set the EnableSsl property to true prior to calling GetResponse(). Similarly, to use SSL with Simple Mail Transport Protocol (SMTP), set the EnableSsl property to true prior to sending the email.

The SslStream class provides a stream-based abstraction for SSL, and offers many ways to configure the SSL handshake.

Example

Code

C#

```
String MyURI = "https://www.contoso.com/";
WebRequest WReq = WebRequest.Create(MyURI);

String serverUri = "ftp://ftp.contoso.com/file.txt"
FtpWebRequest request = (FtpWebRequest)WebRequest.Create(serverUri);
request.EnableSsl = true;
request.Method = WebRequestMethods.Ftp.DeleteFile;
FtpWebResponse response = (FtpWebResponse)request.GetResponse();
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

See also

- Security in Network Programming
- Network programming in .NET
- Certificate Selection and Validation

Certificate Selection and Validation

Article • 08/27/2022

The System.Net classes support several ways to select and validate System.Security.Cryptography.X509Certificates for Secure Socket Layer (SSL) connections. A client can select one or more certificates to authenticate itself to a server. A server can require that a client certificate have one or more specific attributes for authentication.

Definition

A certificate is an ASCII byte stream that contains a public key, attributes (such as version number, serial number, and expiration date) and a digital signature from a Certificate Authority. Certificates are used to establish an encrypted connection or to authenticate a client to a server.

Client Certificate Selection and Validation

A client can select one or more certificates for a specific SSL connection. Client certificates can be associated with the SSL connection to a web server or an SMTP mail server. A client adds certificates to a collection of X509Certificate or X509Certificate2 class objects. Using email as an example, the certificate collection is an instance of a X509CertificateCollection) associated with the ClientCertificates property of the SmtpClient class. The HttpWebRequest class has a similar ClientCertificates property.

The primary difference between the X509Certificate and the X509Certificate2 class is that the private key must reside in the certificate store for the X509Certificate class.

Even if certificates are added to a collection and associated with a specific SSL connection, no certificates will be sent to the server unless the server requests them. If multiple client certificates are set on a connection, the best one will be used based on an algorithm that considers the match between the list of certificate issuers provided by the server and the client certificate issuer name.

The SslStream class provides even more control over the SSL handshake. A client can specify a delegate to pick which client certificate to use.

A remote server can verify that a client certificate is valid, current, and signed by the appropriate Certificate Authority. A delegate can be added to the ServerCertificateValidationCallback to enforce certificate validation.

Client Certificate Selection

The .NET Framework selects the client certificate to present to the server in the following manner:

1. If a client certificate was presented previously to the server, the certificate is cached when first presented and is reused for subsequent client certificate requests.

2. If a delegate is present, always use the result from the delegate as the client certificate to select. Try to use a cached certificate when possible, but do not use cached anonymous credentials if the delegate has returned null and the certificate collection is not empty.

3. If this is the first challenge for a client certificate, the Framework enumerates the certificates in X509Certificate or the X509Certificate2 class objects associated with the connection, looking for a match between the list of certificate issuers provided by the server and the client certificate issuer name. The first certificate that matches is sent to the server. If no certificate matches or the certificate collection is empty, then an anonymous credential is sent to the server.

Tools for Certificate Configuration

A number of tools are available for client and server certificate configuration.

The *Winhttpcertcfg.exe* tool can be used to configure client certificates. The *Winhttpcertcfg.exe* tool is provided as one of the tools with the Windows Server 2003 Resource Kit. This tool is also available as a download as part of the Windows Server 2003 Resource Kit Tools at www.microsoft.com.

The *HttpCfg.exe* tool can be used to configure server certificates for the HttpListener class. The *HttpCfg.exe* tool is provided as one of the support tools for Windows Server 2003 and Windows XP Service Pack 2. *HttpCfg.exe* and the other support tools are not installed by default on either Windows Server 2003 or Windows XP. On Windows Server 2003. the support tools are installed separately from the following folder and file on the Windows Server 2003 CD-ROM:

\Support\Tools\Suptools.msi

For use with Windows XP Service Pack 2, the Windows XP Support Tools are available as a download from www.microsoft.com.

The source code to a version of the *HttpCfg.exe* tool is also provided as a sample with the Windows Server SDK. The source code to the *HttpCfg.exe* sample is installed by

default with the networking samples as part of the Windows SDK under the following folder:

C:\Program Files\Microsoft SDKs\Windows\v1.0\Samples\NetDS\http\serviceconfig

In addition to these tools, the X509Certificate and X509Certificate2 classes provides methods for loading a certificate from the file system.

See also

- Security in Network Programming
- Network programming in .NET

Internet Authentication

Article • 09/15/2021

The System.Net classes support a variety of client authentication mechanisms, including the standard Internet authentication methods basic, digest, negotiate, NTLM, and Kerberos authentication, as well as custom methods that you can create.

Authentication credentials are stored in the NetworkCredential and CredentialCache classes, which implement the ICredentials interface. When one of these classes is queried for credentials, it returns an instance of the **NetworkCredential** class. The authentication process is managed by the AuthenticationManager class, and the actual authentication process is performed by an authentication module class that implements the IAuthenticationModule interface. You must register a custom authentication module with the **AuthenticationManager** before it can be used; modules for the basic, digest, negotiate, NTLM, and Kerberos authentication methods are registered by default.

NetworkCredential stores a set of credentials associated with a single Internet resource identified by a URI and returns them in response to any call to the GetCredential method. The **NetworkCredential** class is typically used by applications that access a limited number of Internet resources or by applications that use the same set of credentials in all cases.

The **CredentialCache** class stores a collection of credentials for various Web resources. When the GetCredential method is called, **CredentialCache** returns the proper set of credentials, as determined by the URI of the Web resource and the requested authentication scheme. Applications that use a variety of Internet resources with different authentication schemes benefit from using the **CredentialCache** class, since it stores all the credentials and provides them as requested.

When an Internet resource requests authentication, the WebRequest.GetResponse method sends the WebRequest to the **AuthenticationManager** along with the request for credentials. The request is then authenticated according to the following process:

1. The **AuthenticationManager** calls the Authenticate method on each of the registered authentication modules in the order they were registered. The **AuthenticationManager** uses the first module that does not return **null** to carry out the authentication process. The details of the process vary depending on the type of authentication module involved.

2. When the authentication process is complete, the authentication module returns an Authorization to the **WebRequest** that contains the information needed to access the Internet resource.

Some authentication schemes can authenticate a user without first making a request for a resource. An application can save time by preauthenticating the user with the resource, thus eliminating at least one round trip to the server. Or, it can perform authentication during program startup in order to be more responsive to the user later. Authentication schemes that can use preauthentication set the PreAuthenticate property to **true**.

See also

- Basic and Digest Authentication
- NTLM and Kerberos Authentication
- Security in Network Programming

Basic and Digest Authentication

Article • 09/15/2021

The System.Net implementation of basic and digest authentication complies with RFC2617 – HTTP Authentication: Basic and Digest Authentication (available on the World Wide Web Consortium's website).

To use basic and digest authentication, an application must provide a user name and password in the Credentials property of the WebRequest object that it uses to request data from the Internet, as shown in the following example.

C#

```
String MyURI = "http://www.contoso.com/";
WebRequest WReq = WebRequest.Create(MyURI);
WReq.Credentials = new NetworkCredential(UserName, SecurelyStoredPassword);
```

> **⊗ Caution**
>
> Data sent with Basic and Digest Authentication is not encrypted, so the data can be seen by an adversary. Additionally, Basic Authentication credentials (user name and password) are sent in the clear and can be intercepted.

See also

- NTLM and Kerberos Authentication
- Internet Authentication

NTLM and Kerberos Authentication

Article • 10/07/2022

Default NTLM authentication and Kerberos authentication use the Microsoft Windows user credentials associated with the calling application to attempt authentication with the server. When using non-default NTLM authentication, the application sets the authentication type to NTLM and uses a NetworkCredential object to pass the user name, password, and domain to the host, as shown in the following example.

C#

```
string myUri = "http://www.contoso.com/";
using HttpClientHandler handler = new()
{
    Credentials = new NetworkCredential(UserName, SecurelyStoredPassword,
Domain),
};
using HttpClient client = new(handler);
string result = await client.GetStringAsync(myUri);
// Do Other Stuff...
```

Applications that need to connect to Internet services using the credentials of the application user can do so with the user's default credentials, as shown in the following example.

C#

```
string myUri = "http://www.contoso.com/";
using HttpClientHandler handler = new()
{
    Credentials = CredentialCache.DefaultCredentials,
};
using HttpClient client = new(handler);
string result = await client.GetStringAsync(myUri);
// Do Other Stuff...
```

The negotiate authentication module determines whether the remote server is using NTLM or Kerberos authentication, and sends the appropriate response.

> ⓘ **Note**
>
> NTLM authentication does not work through a proxy server.

See also

- Basic and Digest Authentication
- Internet Authentication

Web and Socket Permissions

Article • 09/15/2021

Internet security for applications using the System.Net namespace is provided by the WebPermission and SocketPermission classes. The **WebPermission** class controls an application's right to request data from a URI or to serve a URI to the Internet. The **SocketPermission** class controls an application's right to use a Socket to accept data on a local port or to contact remote devices using a transport protocol at another address, based on the host, port number, and transport protocol of the socket.

Which permission class you use depends on your application type. Applications that use WebRequest and its descendants should use the **WebPermission** class to manage permissions. Applications that use socket-level access should use the **SocketPermission** class to manage permissions.

WebPermission and **SocketPermission** define two permissions: accept and connect. Accept grants the application the right to answer an incoming connection from another party. Connect grants the application the right to initiate a connection to another party.

For **SocketPermission** instances, accept means that an application can accept incoming connections on a local transport address; connect means that an application can connect to some remote (or local) transport address.

For **WebPermission** instances, accept means that an application can export the URI controlled by the **WebPermission** to the world; connect means that an application can access that URI (whether it is remote or local).

See also

- Security
- Security in Network Programming

Best practices for System.Net classes

Article • 08/27/2022

The following recommendations will help you use the classes contained in System.Net to their best advantage:

- For Transport Layer Security (TLS) best practices, see Transport Layer Security (TLS) best practices with .NET Framework.
- Use HttpClient to send HTTP requests instead of WebRequest, which was obsoleted in .NET 6. In .NET Framework, create a new `HttpClient` instance each time you need to send a request. (The guidance for .NET 5+/.NET Core is more nuanced. For more information, see Guidelines for using HttpClient.)
- When writing ASP.NET applications that run on a server using the `System.Net` classes, it's often better, from a performance standpoint, to use the asynchronous method SendAsync instead of Send.
- The number of connections opened to an internet resource can have a significant impact on network performance and throughput. **System.Net** uses two connections per application per host by default. Setting the ConnectionLimit property in the ServicePoint for your application can increase this number for a particular host. Setting the ServicePointManager.DefaultPersistentConnectionLimit property can increase this default for all hosts.
- When writing socket-level protocols, try to use TcpClient or UdpClient whenever possible instead of writing directly to a Socket. These two client classes encapsulate the creation of TCP and UDP sockets without requiring you to handle the details of the connection.
- When accessing sites that require credentials, use the CredentialCache class to create a cache of credentials rather than supplying them with every request. The **CredentialCache** class searches the cache to find the appropriate credential to present with a request, relieving you of the responsibility of creating and presenting credentials based on the URL.

See also

- Network programming in .NET

Peer Name Resolution Protocol

Article • 08/27/2022

In peer-to-peer environments, peers use specific name resolution systems to resolve each other's network locations (addresses, protocols, and ports) from names or other types of identifiers. In the past, peer name resolution has been complicated by the inherently transient connectivity as well as other shortcomings within the Domain Name System (DNS).

The Microsoft® Windows® Peer-to-Peer Networking platform solves this problem with the Peer Name Resolution Protocol (PNRP), a secure, scalable, and dynamic name registration and name resolution protocol first developed for Windows XP and then upgraded in Windows Vista™. PNRP works very differently from traditional name resolution systems, opening up exciting new possibilities for application developers.

With PNRP, peer names can be applied to the machine, or individual applications or services on the machine. A peer name resolution includes an address, port, and possibly an extended payload. Benefits of this system include fault tolerance, no bottlenecks, and name resolutions that will never return stale addresses; making the protocol an excellent solution for locating mobile users.

In terms of security, peer names can be published as secured (protected) or unsecured (unprotected). PNRP uses public key cryptography to protect secure peer names against spoofing; both computers and services can be named with PNRP.

The Peer Name Resolution Protocol demonstrates the following properties:

- Distributed and almost entirely serverless. Servers are only required for the bootstrapping process.
- Secure name publication without the involvement of third parties. Unlike DNS name publication, PNRP name publication is instantaneous and without financial cost.
- PNRP updates in real-time, which prevents the resolution of stale addresses.
- The resolution of names via PNRP extends beyond computers by also allowing name resolution for services.

The System.Net.PeerToPeer namespace

- PNRP functionality is defined by the System.Net.PeerToPeer namespace within .NET Framework version 3.5. It provides a set of types that can be used to register and resolve peer names with an available PNRP service.
- (PNRP and custom peer resolvers can be created and instantiated using the types provided in the System.ServiceModel.PeerResolvers namespace.)
- The basic types used to register and resolve names with an available PNRP service are as follows:
- Cloud: Defines the information describing an available PNRP cloud, including its scope.
- PeerName: Defines a peer name that can be used to register and subsequently resolve a peer within a cloud.
- PeerNameRecord: Defines the record in PNRP cloud that contains the registration information for a peer, which includes the network endpoints at which the peer can be contacted.
- PeerNameRegistration: Defines the registration process for a peer name, including methods to start and stop peer name registration.
- PeerNameResolver: Defines the process for resolving a peer name to its network endpoint(s), including both synchronous and asynchronous methods for resolution.

See also

- System.ServiceModel.PeerResolvers
- System.Net.PeerToPeer

Peer Names and PNRP IDs

Article • 09/15/2021

A Peer Name represents an endpoint for communication, which can be a computer, a user, a group, a service, or anything associated with a Peer that can be resolved to an IPv6 address. The Peer Name Resolution Protocol (PNRP) takes the statistically unique Peer Name for the creation of a PNRP ID, which is used to identify cloud members.

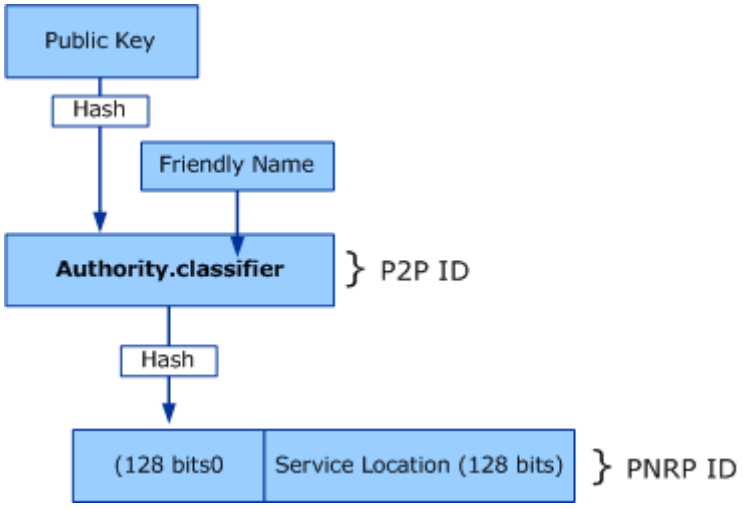
Peer Names

Peer names can be registered as unsecured or secured. Unsecured names are just text strings that are subject to spoofing, as anyone can register a duplicate unsecured name. Unsecured names are best used in private or otherwise protected networks. Secured names are protected with a certificate and a digital signature. Only the original publisher will be able to prove ownership of a secured name.

The combination of cloud and scope provides a reasonably secure environment for peers that participate in PNRP activity. However, using a secured peer name does not ensure the overall security of the networking application. Security of the application is implementation-dependent.

Secured peer names are only registered by their owner and are protected with public key cryptography. A secured peer name is considered owned by the peer entity having the corresponding private key. Ownership can be proved via the certified peer address (CPA), which is signed using the private key. A malicious user cannot forge ownership of a peer name without the corresponding private key.

PNRP IDs



PNRP IDs are composed of the following:

- The high-order 128 bits, known as the peer-to-peer (P2P) ID, are a hash of a peer name assigned to the endpoint. The peer name has the following format: *Authority.Classifier*. For secured names, *Authority* is the Secure Hash Algorithm 1 (SHA1) hash of the public key of the peer name in hexadecimal characters. For unsecured names, the *Authority* is the single character "0". *Classifier* is a string that identifies the application. No peer name classifier can be greater than 149 characters long, including the `null` terminator.

- The low-order 128 bits are used for the Service Location, which is a generated number that identifies different instances of the same P2P ID in the same cloud.

This combination of P2P ID and Service Location allows multiple PNRP IDs to be registered from a single computer.

See also

- PeerName
- System.Net.PeerToPeer

Peer Name Publication and Resolution

Article • 09/15/2021

Publishing a peer name

To publish a new PNRP ID, a peer performs the following:

- Sends PNRP publication messages to its cache neighbors (the peers that have registered PNRP IDs in the lowest level of the cache) to seed their caches.
- Chooses random nodes in the cloud that are not its neighbors and sends them PNRP name resolution requests for its own P2P ID. The resulting endpoint determination process seeds the caches of random nodes in the cloud with the PNRP ID of the publishing peer.

PNRP version 2 nodes do not publish PNRP IDs if they are only resolving other P2P IDs. The HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\PeerNet\PNRP\IPV6-Global\SearchOnly=1 registry value (REG_DWORD type) specifies that peers only use PNRP for name resolution, never for name publication. This registry value can also be configured through Group Policy.

Resolving a peer name

Locating other peers in a PNRP network or cloud is a process comprised of two phases:

1. Endpoint Determination
2. PNRP ID Resolution

In the endpoint determination phase, a peer that is attempting to resolve the PNRP ID of a service on another computer determines the IPv6 address of that remote peer. The remote peer is the one that published, or is associated with, the PNRP ID of the computer or service.

After confirming that the remote endpoint has been registered into the PNRP cloud, the requesting peer in the PNRP ID resolution phase sends a request to that peer endpoint for the PNRP ID of the desired service. The endpoint sends a reply confirming the PNRP ID of the service, a comment, and up to 4 kilobytes of additional information that the requesting peer can use for future communication. For example, if the desired endpoint is a gaming server, the additional peer name record data can contain information about the game, the level of play, and the current number of players.

In the endpoint determination phase, PNRP uses an iterative process for locating the node that published the PNRP ID, in which the node performing the resolution is responsible for contacting nodes that are successively closer to the target PNRP ID.

To perform name resolution in PNRP, the peer examines the entries in its own cache for an entry that matches the target PNRP ID. If found, the peer sends a PNRP Request message to the peer and waits for a response. If an entry for the PNRP ID is not found, the peer sends a PNRP Request message to the peer that corresponds to the entry that has a PNRP ID that most closely matches the target PNRP ID. The node that receives the PNRP Request message examines its own cache and does the following:

- If the PNRP ID is found, the requested endpoint peer replies directly to the requesting peer.
- If the PNRP ID is not found and a PNRP ID in the cache is closer to the target PNRP ID, the requested peer sends a response to the requesting peer containing the IPv6 address of the peer that represents the entry with a PNRP ID that more closely matches the target PNRP ID. Using the IP address in the response, the requesting node sends another PNRP Request message to the IPv6 address to respond or examine its cache.
- If the PNRP ID is not found and there is no PNRP ID in its cache that is closer to the target PNRP ID, the requested peer sends the requesting peer a response that indicates this condition. The requesting peer then chooses the next-closest PNRP ID.

The requesting peer continues this process with successive iterations, eventually locating the node that registered the PNRP ID.

Within the System.Net.PeerToPeer namespace, there is a many-to-many relationship between the PeerName records that contain endpoints and PNRP clouds or meshes in which they communicate. When there are duplicate or stale entries, or multiple nodes with the same peer name, PNRP nodes can obtain current information using the PeerNameResolver class. The PeerNameResolver methods use a single peer name to simplify the perspective to one peer-to-many peer name records and the same one peer to many clouds. This is similar to a query performed using a relational-table join. Upon successful completion, the Resolver object returns a PeerNameRecordCollection for the specified peer name. For example, a peer name would occur in all the peer name records in the collection, ordered by cloud. These are the instances of the peer name whose supporting data may be requested by a PNRP-based application.

See also

- System.Net.PeerToPeer

PNRP Clouds

Article • 09/15/2021

A PNRP "cloud" represents a set of nodes that can communicate with each other through the network. The term "cloud" is synonymous with "peer mesh" and "peer-to-peer graph".

Communication between nodes should never cross from one cloud to another. A Cloud instance is uniquely identified by its name, which is case-sensitive. A single peer or node may be connected to more than one cloud.

Clouds are tied very closely to network interfaces. On a multi-homed machine with two network cards attached to different subnets, three clouds will be returned: one for each of the link local addresses per interface, and a single global scope cloud.

PNRP uses three cloud "scopes", in which a scope is a grouping of computers that are able to find each other:

- The global cloud corresponds to the global IPv6 address scope and global addresses and represents all the computers on the entire IPv6 Internet. There is only a single global cloud.
- The link-local cloud corresponds to the link-local IPv6 address scope and link-local addresses. A link-local cloud is for a specific link, which is typically the same as the locally attached subnet. There can be multiple link-local clouds.

A third cloud, the site-specific cloud, corresponds to the site IPv6 address scope and site-local addresses. This cloud has been deprecated, although it is still supported in PNRP.

Clouds

PNRP clouds are represented by instances of the Cloud class. Groups of clouds used a peer are represented by instances of the enumerable CloudCollection class. Collections of PNRP clouds known to the current peer can be obtained by calling the static GetAvailableClouds method.

Individual clouds have unique names, represented as a 256 character Unicode string. These names, along with the above-mentioned scope, are used to construct unique instances of the Cloud class. These instances can be serialized and reconstructed for persistent usage.

Once a Cloud instance is created or obtained, peer names can be registered with it to create a mesh of known peers.

See also

- Cloud
- Peer Name Resolution Protocol

PNRP Caches

Article • 09/15/2021

Peer Name Resolution Protocol (PNRP) caches are local collections of algorithmically selected peer endpoints maintained on the peer.

PNRP Cache Initialization

To initialize the PNRP cache, or Peer Name Record Collection, when a peer node starts up, a node can use the following methods:

- Persistent cache entries that were present when the node was shut down are loaded from hard disk storage.
- If an application uses the P2P collaboration infrastructure, collaboration information is available in the Contact Manager for that node.

Scaling Peer Name Resolution with a Multi-Level Cache

To keep the sizes of the PNRP caches small, peer nodes use a multi-level cache, in which each level contains a maximum number of entries. Each level in the cache represents a one tenth smaller portion of the PNRP ID number space (2^{256}). The lowest level in the cache contains a locally registered PNRP ID and other PNRP IDs that are numerically close to it. As a level of the cache is filled with a maximum of 20 entries, a new lower level is created. The maximum number of levels in the cache is on the order of log10(Total number of PNRP IDs in the cloud). For example, for a global cloud with 100 million PNRP IDs, there are no more than 8 (=log10(100,000,000)) levels in the cache and a similar number of hops to resolve a PNRP ID during name resolution. This mechanism allows for a distributed hash table for which an arbitrary PNRP ID can be resolved by forwarding PNRP Request messages to the next-closest peer until the peer with the corresponding CPA is found.

To ensure that resolution can complete, each time a node adds an entry to the lowest level of its cache, it floods a copy of the entry to all the nodes within the last level of the cache.

The cache entries are refreshed over time. Cache entries that are stale are removed from the cache. The result is that the distributed hash table of PNRP IDs is based on active

endpoints, unlike DNS in which address records and the DNS protocol provide no guarantee that the node associated with the address is actively on the network.

Other PNRP Caches

Another persistent data store is the local cache. In addition to the other objects needed for PNRP activity, it may include the records associated with a PNRP cloud or collaboration session that is securely published and synchronized between all the members of the cloud. This replicated store represents the view of the group data, which should be the same for all group members. Technically, these objects are not records per se, but rather application, presence, and object data destined for a local cache. Use of the PNRP cloud ensures that objects are propagated to all nodes in the collaboration session or PNRP cloud. Record replication between cloud members uses SSL to provide encryption and data integrity.

When a peer joins a cloud, they do not automatically receive local cache data from the host peer to which they attach; they have to subscribe to the host peer to receive updates in application, presence, and object data. After the initial synchronization, peers periodically resynchronize their replicated stores to ensure that all group members consistently have the same view. The collaboration session or applications within the collaboration session may also perform the same function.

After a collaboration session has begun for a cloud, applications can register peers and begin publishing their information using the security defined by the cloud scope. When a peer joins a cloud, the security mechanisms for the cloud are applied to the peer, giving it a scope in which to participate. Its records can then be published securely within the scope of the cloud. Note that cloud scope may not be the same as collaboration application scope.

Peers can register interest in receiving objects from other peers. When an object is updated, the collaboration application is notified and the new object is passed to all subscribers of the application. For example, a peer in a group chat application can register interest in receiving application information, which will send it all chat records as application data. This allows it to monitor chat activity within the cloud.

See also

- System.Net.PeerToPeer

PNRP in Application Development

Article • 09/15/2021

In Windows Vista, networking applications can access name publication and resolution functions through a simplified PNRP application programming interface (API).

Implementing the Peer Name Resolution Protocol

With the simplified PNRP API, clouds are not explicitly specified to register the name and addresses; the PNRP component automatically determines the appropriate clouds to join and the addresses to publish within the clouds.

For highly simplified PNRP name resolution in Windows Vista, PNRP names are now integrated into the getaddrinfo() Windows Sockets function. To use PNRP to resolve a name to an IPv6 address, applications can use the getaddrinfo() function to resolve the Fully Qualified Domain Name (FQDN) name.prnp.net, in which name is peer name being resolved. The pnrp.net domain is a reserved domain in Windows Vista for PNRP name resolution.

Message passing between PeerToPeer applications is still handled by underlying architectures such as PeerChannel and WCF Large Data and Streaming.

See also

- System.Net.PeerToPeer

Peer-to-peer collaboration

Article • 09/15/2021

Peer-to-peer networking is the utilization of the relatively powerful computers (personal computers) that exist at the edge of the Internet for more than just client-based computing tasks. The modern personal computer (PC) has a very fast processor, vast memory, and a large hard disk, none of which are being fully utilized when performing common computing tasks such as email and Web browsing. The modern PC can easily act as both a client and server (a peer) for many types of applications.

The Peer-to-Peer Collaboration Infrastructure is a simplified implementation of the Microsoft Windows Peer-to-Peer Infrastructure that leverages the People Near Me service in Windows Vista and later platforms. It is best used for peer-enabled applications within a subnet for which the People Near Me service operates, although it can service internet endpoints or contacts as well. It incorporates the common Contact Manager that is used by Live Messenger and other Live-aware applications to determine contact endpoints, availability, and presence.

Collaboration applications

A typical peer-to-peer collaboration application is comprised of the following steps:

- Peer determines the identity of a peer who is interested in hosting a collaboration session
- A request to host a session is sent, somehow, and the host peer agrees to manage collaboration activity.
- The host invites contacts on the subnet (including the requester) to a session.
- All peers who want to collaborate may add the host to their contact managers.
- Most peers will send invitation responses, whether accepted or declined, back to the host peer in a timely fashion.
- All peers who want to collaborate will subscribe to the host peer.
- While the peers are performing their initial collaboration activity, the host peer may add remote peers to its contact manager. It also processes all invitation responses to determine who has accepted, who has declined, and who has not answered. It may cancel invitations to those who have not answered, or perform some other activity.

- At this point, the host peer can start a collaboration session with all invited peers, or register an application with the collaboration infrastructure. P2P applications use the Peer-to-Peer Collaboration Infrastructure and the System.Net.PeerToPeer.Collaboration namespace to coordinate communications for games, bulletin boards, conferencing, and other serverless presence applications.

Peer-to-peer networking security

In an Active Directory domain, domain controllers provide authentication services using Kerberos. In a serverless peer environment, the peers must provide their own authentication. For Peer-to-Peer Networking, any node can act as a CA, removing the requirement of a root certificate in each peer's trusted root store. Authentication is provided using self-signed certificates, formatted as X.509 certificates. These are certificates that are created by each peer, which generates the public key/private key pair and the certificate that is signed using the private key. The self-signed certificate is used for authentication and to provide information about the peer entity. Like X.509 authentication, peer networking authentication relies upon a chain of certificates tracing back to a public key that is trusted.

See also

- System.Net.PeerToPeer.Collaboration
- About the System.Net.PeerToPeer.Collaboration Namespace

About the System.Net.PeerToPeer.Collaboration Namespace

Article • 09/15/2021

The System.Net.PeerToPeer.Collaboration namespace provides classes and APIs that are used to implement peer collaboration activities using the Peer-to-Peer Collaboration Infrastructure.

Classes

The main classes used in the implementation of a Peer-to-Peer Collaboration activity are:

- The ContactManager, which can be used to store peer contacts.
- The PeerApplication in which to collaborate, such as a game, chat client, or conferencing solution.
- The peers that will be collaborating in an activity. These peers can be represented as PeerContact, PeerNearMe, or PeerEndPoint objects.
- The static PeerCollaboration class itself, which specifies which applications are available and which peers are participating in them.

The Invite methods are used to invite peers to a collaboration session. A calling peer can subscribe to another peer for events that signal updates to application, object, or presence information affiliated with the collaboration session. Presence classes specify whether a Peer is available for collaboration, and the PeerScope class is used to specify how much participation is allowed for a peer: Internet (global), NearMe, (subnet) or None.

A collaboration session is comprised of four steps:

- Discovery. Discover or publish applications, peers, and presence information. For instance, find other people on the local subnet that have the same games installed.
- Invitation. Send and accept secure invitations for remote peer(s) to start or join PeerCollaboration sessions.
- Contact Management. Add discovered peers as a contact to a ContactManager.

- Communication. When communication is established, use the System.Net APIs, the System.Net.PeerToPeer API, or the Windows Communication Foundation Peer Channel classes for multiparty communications.

For example, the host peer starts a collaboration session, and utilizes the CreateContact method to add a remote peer and one of its local peers to the Contact Manager of the host peer. The three users will then participate in their own private collaboration session.

Typical P2P applications are: conference calls for collaborative note-taking or whiteboarding, serverless chat applications, interactive advertisements, and online gaming sessions.

See also

- System.Net.PeerToPeer.Collaboration

Peer-to-Peer Networking Scenarios

Article • 09/15/2021

Peer-to-peer networking enables or enhances the following scenarios:

Real-Time Communications (RTC)

- Serverless Instant Messaging

  RTC exists today. Computer users can chat and have voice or video conversations with their peers today. However, many of the existing programs and their communications protocols rely on servers to function. If you are participating in an ad-hoc wireless network or are a part of an isolated network, you are unable to use these RTC facilities. Peer-to-peer technology allows the extension of RTC technologies to these additional networking environments.

- Real-time matchmaking and gameplay

  Similar to RTC, real-time game play exists today. There are many Web-based game sites that cater to the gaming community via the Internet. They offer the ability to find other gamers with similar interests and play a game together. The problem is that the game sites exist only on the Internet and are geared toward the avid gamer who wants to play against the best gamers in the world. These sites track and provide the statistics to help in the process. However, these sites do not allow a gamer to set up an ad-hoc game among friends in a variety of networking environments. Peer-to-peer networking can provide this capability.

Collaboration

- Project workspaces solving a goal

  Shared workspace applications allow for the creation of ad-hoc workgroups and then allow the workgroup owners to populate the shared workspace with the tools and content that will allow the group to solve a problem. This could include message boards, productivity tools, and files.

- Sharing files with others

  A subset of project workspace sharing is the ability to share files. Although this ability exists today with the current version of Windows, it can be enhanced through peer-to-peer networking to make file content available in an easy and

friendly way. Allowing easy access to the incredible wealth of content at the edge of the Internet or in ad-hoc computing environments increases the value of network computing.

- Sharing experiences

  With wireless connectivity becoming more prevalent, peer-to-peer networking allows you to be online in a group of peers and to be able to share your experiences (such as a sunset, a rock concert, or a vacation cruise) while they are occurring.

Content Distribution

- Text messages

  Peer-to-peer networking can allow for the dissemination of text-based information in the form of files or messages to a large group of users. An example is a news list.

- Audio and video

  Peer-to-peer networking can also allow for the dissemination of audio or video information to a large group of users, such as a large concert or company meeting. To distribute the content today, you must configure high-capacity servers to collect and distribute the load to hundreds or thousands of users. With peer-to-peer networking, only a handful of peers would actually get their content from the centralized servers. These peers would flood this information out to a few more people who send it to others, and so on. The load of distributing the content is distributed to the peers in the cloud. A peer that wants to receive the content would find the closest distributing peer and get the content from them.

- Distribution of product updates

  Peer-to-peer networking can also provide an efficient mechanism to distribute software such as product updates (security updates and service packs). A peer that has a connection to a software distribution server can obtain the product update and propagate it to the other members of its group.

Distributed Processing

- Division and distribution of a task

A large computing task can first be divided into separate smaller computing tasks well suited to the computing resources of a peer. A peer could do the dividing of the large computing task. Then, peer-to-peer networking can distribute the individual tasks to the separate peers in the group. Each peer performs its computing task and reports its result back to a centralized accumulation point.

- Aggregation of computer resources

  Another way to utilize peer-to-peer networking for distributed processing is to run programs on each peer that run during idle processor times and are part of a larger computing task that is coordinated by a central server. By aggregating the processors of multiple computers, peer-to-peer networking can turn a group of peer computers into a large parallel processor for large computing tasks.

See also

- System.Net.PeerToPeer.Collaboration

Integrated Windows Authentication with Extended Protection

Article • 09/15/2021

Enhancements were made that affect how integrated Windows authentication is handled by the HttpWebRequest, HttpListener, SmtpClient, SslStream, NegotiateStream, and related classes in the System.Net and related namespaces. Support was added for extended protection to enhance security.

These changes can affect applications that use these classes to make web requests and receive responses where integrated Windows authentication is used. This change can also impact web servers and client applications that are configured to use integrated Windows authentication.

These changes can also affect applications that use these classes to make other types of requests and receive responses where integrated Windows authentication is used.

The changes to support extended protection are available only for applications on Windows 7 and Windows Server 2008 R2. The extended protection features are not available on earlier versions of Windows.

Overview

The design of integrated Windows authentication allows for some credential challenge responses to be universal, meaning they can be re-used or forwarded. The challenge responses should be constructed at a minimum with target specific information and preferably also with some channel specific information. Services can then provide extended protection to ensure that credential challenge responses contain service specific information such as a Service Principal Name (SPN). With this information in the credential exchanges, services are able to better protect against malicious use of credential challenge responses that might have been improperly used.

The extended protection design is an enhancement to authentication protocols designed to mitigate authentication relay attacks. It revolves around the concept of channel and service binding information.

The overall objectives are the following:

1. If the client is updated to support the extended protection, applications should supply a channel binding and service binding information to all supported authentication protocols. Channel binding information can only be supplied when

there is a channel (TLS) to bind to. Service binding information should always be supplied.

2. Updated servers which are properly configured may verify the channel and service binding information when it is present in the client authentication token and reject the authentication attempt if the channel bindings do not match. Depending on the deployment scenario, servers may verify channel binding, service binding or both.

3. Updated servers have the ability to accept or reject down-level client requests that do not contain the channel binding information based on policy.

Information used by extended protection consists of one or both of the following two parts:

1. A Channel Binding Token or CBT.

2. Service Binding information in the form of a Service Principal Name or SPN.

Service Binding information is an indication of a client's intent to authenticate to a particular service endpoint. It is communicated from client to server with the following properties:

- The SPN value must be available to the server performing client authentication in clear text form.

- The value of the SPN is public.

- The SPN must be cryptographically protected in transit such that a man-in-the-middle attack cannot insert, remove or modify its value.

A CBT is a property of the outer secure channel (such as TLS) used to tie (bind) it to a conversation over an inner, client-authenticated channel. The CBT must have the following properties (also defined by IETF RFC 5056):

- When an outer channel exists, the value of the CBT must be a property identifying either the outer channel or the server endpoint, independently arrived at by both client and server sides of a conversation.

- Value of the CBT sent by the client must not be something an attacker can influence.

- No guarantees are made about secrecy of the CBT value. This does not however mean that the value of the service binding as well as channel binding information

can always be examined by any other but the server performing authentication, as the protocol carrying the CBT may be encrypting it.

- The CBT must be cryptographically integrity protected in transit such that an attacker cannot insert, remove or modify its value.

Channel binding is accomplished by the client transferring the SPN and the CBT to the server in a tamperproof fashion. The server validates the channel binding information in accordance with its policy and rejects authentication attempts for which it does not believe itself to have been the intended target. This way, the two channels become cryptographically bound together.

To preserve compatibility with existing clients and applications, a server may be configured to allow authentication attempts by clients that do not yet support extended protection. This is referred to as a "partially hardened" configuration, in contrast to a "fully hardened" configuration.

Multiple components in the System.Net and System.Net.Security namespaces perform integrated Windows authentication on behalf of a calling application. This section describes changes to System.Net components to add extended protection in their use of integrated Windows authentication.

Extended protection is currently supported on Windows 7. A mechanism is provided so an application can determine if the operating system supports extended protection.

Changes to Support Extended Protection

The authentication process used with integrated Windows authentication, depending on the authentication protocol used, often includes a challenge issued by the destination computer and sent back to the client computer. Extended protection adds new features to this authentication process

The System.Security.Authentication.ExtendedProtection namespace provides support for authentication using extended protection for applications. The ChannelBinding class in this namespace represents a channel binding. The ExtendedProtectionPolicy class in this namespace represents the extended protection policy used by the server to validate incoming client connections. Other class members are used with extended protection.

For server applications, these classes include the following:

A ExtendedProtectionPolicy that has the following elements:

- An OSSupportsExtendedProtection property that indicates whether the operating system supports integrated windows authentication with extended protection.

- A PolicyEnforcement value that indicates when the extended protection policy should be enforced.
- A ProtectionScenario value that indicates the deployment scenario. This influences how extended protection is checked.
- An optional ServiceNameCollection that contains the custom SPN list that is used to match against the SPN provided by the client as the intended target of the authentication.
- An optional ChannelBinding that contains a custom channel binding to use for validation. This scenario is not a common case

The System.Security.Authentication.ExtendedProtection.Configuration namespace provides support for configuration of authentication using extended protection for applications.

A number of feature changes were made to support extended protection in the existing System.Net namespace. These changes include the following:

- A new TransportContext class added to the System.Net namespace that represents a transport context.
- New EndGetRequestStream and GetRequestStream overload methods in the HttpWebRequest class that allow retrieving the TransportContext to support extended protection for client applications.
- Additions to the HttpListener and HttpListenerRequest classes to support server applications.

A feature change was made to support extended protection for SMTP client applications in the existing System.Net.Mail namespace:

- A TargetName property in the SmtpClient class that represents the SPN to use for authentication when using extended protection for SMTP client applications.

A number of feature changes were made to support extended protection in the existing System.Net.Security namespace. These changes include the following:

- New BeginAuthenticateAsClient and AuthenticateAsClient overload methods in the NegotiateStream class that allow passing a CBT to support extended protection for client applications.
- New BeginAuthenticateAsServer and AuthenticateAsServer overload methods in the NegotiateStream class that allow passing an ExtendedProtectionPolicy to

support extended protection for server applications.

- A new TransportContext property in the SslStream class to support extended protection for client and server applications.

A SmtpNetworkElement property was added to support configuration of extended protection for SMTP clients in the System.Net.Security namespace.

Extended Protection for Client Applications

Extended protection support for most client applications happens automatically. The HttpWebRequest and SmtpClient classes support extended protection whenever the underlying version of Windows supports extended protection. An HttpWebRequest instance sends an SPN constructed from the Uri. By default, an SmtpClient instance sends an SPN constructed from the host name of the SMTP mail server.

For custom authentication, client applications can use the HttpWebRequest.EndGetRequestStream(IAsyncResult, TransportContext) or HttpWebRequest.GetRequestStream(TransportContext) methods in the HttpWebRequest class that allow retrieving the TransportContext and the CBT using the GetChannelBinding method.

The SPN to use for integrated Windows authentication sent by an HttpWebRequest instance to a given service can be overridden by setting the CustomTargetNameDictionary property.

The TargetName property can be used to set a custom SPN to use for integrated Windows authentication for the SMTP connection.

Extended Protection for Server Applications

HttpListener automatically provides mechanisms for validating service bindings when performing HTTP authentication.

The most secure scenario is to enable extended protection for `HTTPS://` prefixes. In this case, set HttpListener.ExtendedProtectionPolicy to an ExtendedProtectionPolicy with PolicyEnforcement set to WhenSupported or Always, and ProtectionScenario set to TransportSelected A value of WhenSupported puts HttpListener in partially hardened mode, while Always corresponds to fully hardened mode.

In this configuration when a request is made to the server through an outer secure channel, the outer channel is queried for a channel binding. This channel binding is

passed to the authentication SSPI calls, which validate that the channel binding in the authentication blob matches. There are three possible outcomes:

1. The server's underlying operating system does not support extended protection. The request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

2. The SSPI call fails indicating that either the client specified a channel binding that did not match the expected value retrieved from the outer channel or the client failed to supply a channel binding when the extended protection policy on the server was configured for Always. In both cases, the request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

3. The client specifies the correct channel binding or is allowed to connect without specifying a channel binding since the extended protection policy on the server is configured with WhenSupported The request is returned to the application for processing. No service name check is performed automatically. An application may choose to perform its own service name validation using the ServiceName property, but under these circumstances it is redundant.

If an application makes its own SSPI calls to perform authentication based on blobs passed back and forth within the body of an HTTP request and wishes to support channel binding, it needs to retrieve the expected channel binding from the outer secure channel using HttpListener in order to pass it to native Win32 AcceptSecurityContext function. To do this, use the TransportContext property and call GetChannelBinding method to retrieve the CBT. Only endpoint bindings are supported. If anything other Endpoint is specified, a NotSupportedException will be thrown. If the underlying operating system supports channel binding, the GetChannelBinding method will return a ChannelBindingSafeHandle wrapping a pointer to a channel binding suitable for passing to AcceptSecurityContext function as the pvBuffer member of a SecBuffer structure passed in the `pInput` parameter. The Size property contains the length, in bytes, of the channel binding. If the underlying operating system does not support channel bindings, the function will return `null`.

Another possible scenario is to enable extended protection for `HTTP://` prefixes when proxies are not used. In this case, set HttpListener.ExtendedProtectionPolicy to an ExtendedProtectionPolicy with PolicyEnforcement set to WhenSupported or Always, and ProtectionScenario set to TransportSelected A value of WhenSupported puts

HttpListener in partially hardened mode, while Always corresponds to fully hardened mode.

A default list of allowed service names is created based on the prefixes which have been registered with the HttpListener. This default list can be examined through the DefaultServiceNames property. If this list is not comprehensive, an application can specify a custom service name collection in the constructor for the ExtendedProtectionPolicy class which will be used instead of the default service name list.

In this configuration, when a request is made to the server without an outer secure channel authentication proceeds normally without a channel binding check. If the authentication succeeds, the context is queried for the service name that the client provided and validated against the list of acceptable service names. There are four possible outcomes:

1. The server's underlying operating system does not support extended protection. The request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

2. The client's underlying operating system does not support extended protection. In the WhenSupported configuration, the authentication attempt will succeed and the request will be returned to the application. In the Always configuration, the authentication attempt will fail. The request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

3. The client's underlying operating system supports extended protection, but the application did not specify a service binding. The request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

4. The client specified a service binding. The service binding is compared to the list of allowed service bindings. If it matches, the request is returned to the application. Otherwise, the request will not be exposed to the application, and an unauthorized (401) response will be automatically returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

If this simple approach using an allowed list of acceptable service names is insufficient, an application may provide its own service name validation by querying the ServiceName property. In cases 1 and 2 above, the property will return `null`. In case 3, it

will return an empty string. In case 4, the service name specified by the client will be returned.

These extended protection features can also be used by server applications for authentication with other types of requests and when trusted proxies are used.

See also

- System.Security.Authentication.ExtendedProtection
- System.Security.Authentication.ExtendedProtection.Configuration