**[Certificate based Authentication and WCF (Message Security)](http://www.leastprivilege.com/CertificateBasedAuthenticationAndWCFMessageSecurity.aspx)**

When using message security, the intended way to validate an incoming credential (== token) is a token validator. You can find several internal validators in the *System.IdentityModel.Selectors* namespace (e.g. for UserName, X.509 or Windows tokens). The X509 token validators gets called whenever an incoming certificate has to be validated - when you have secure conversation enabled, this happens only on the first request which makes this approach very efficient.

WCF has three builtin validators for X.509 certificates and you can choose which one to use via a service/endpoint behavior. I will show the service side settings here, but the same switches also exist on the client side.

<behaviors>

<serviceBehaviors>

<behavior name="behavior">

<serviceCredentials>

<clientCertificate>

<authentication certificateValidationMode="ChainTrust"

revocationMode="Online" />

</clientCertificate>

</serviceCredentials>

</behavior>

</serviceBehaviors>

</behaviors>

The *certificateValidationMode* specifies how incoming certificates are validated and how trust is determined:

* **None**. No validation is performed. Not recommended.
* **ChainTrust**. The certificate has to chain up to one of the CAs in your trusted CA certificate folder.
* **PeerTrust**. The incoming certificate has to be in the Trusted People certificate folder.
* **PeerOrChainTrust**. A combination of Chain and Peer trust.
* **Custom**. For all other cases.

The *trustedStoreLocation* attribute determines whether the current user or local machine store is used for peer or chain checks - defaults to local machine. Furthermore you specify how revocation lists should get checked via the *revocationMode* attribute (no check, offline or online).

See this [post](http://www.leastprivilege.com/CertificateBasedAuthenticationAndWCF.aspx) for a more detailed description of the validation modes. Also, as mentioned in that post, the standard validation modes are mostly useful in niche situations.

This is where the *Custom* mode comes into play. With this mode it is your responsibility to validate the certificate following your own guidelines. You then make decisions if you want to accept the certificate.

A custom certificate validator involves deriving from *X509CertificateValidator* and implementing the *Validate()* method. The WCF plumbing passes the incoming certificate into this method. If you want to reject the certificate you throw a *SecurityTokenValidationException* inside *Validate()*.

So far so good - now, which steps are involved to validate a certificate? Before you can rely on any information in the cert, you have to make sure it's valid - typcially by checking it is not expired or revoked and is issued by a trusted CA. Afterwards you can check certain properties of the certificate or its issuer to further restrict the allowed certs. Another approach would be - like *PeerTrust* - to check the certificate against a list of allowed certificates.

For checking the trust chain, you use the *X509Chain* class - the general logic of Validate() looks like this:

public override void Validate(X509Certificate2 certificate)

{

// create chain and set validation options

X509Chain chain = new X509Chain();

SetValidationSettings(chain);

// check if cert is valid and chains up to a trusted CA

if (!chain.Build(certificate))

{

throw new SecurityTokenValidationException(  
 "Client certificate is not valid");

}

// check if cert is from our trusted list

if (!IsTrusted(chain, GetTrustedThumbprints()))

{

throw new SecurityTokenValidationException(  
 "Client certificate is not trusted");

}

}

How the certificate should be exactly validated can be specified on the *ChainPolicy* property of *X509Chain*. Besides the configurable revocation mode, WCF uses the default settings for the chain policy, which are:

protected override void SetValidationSettings(X509Chain chain)

{

chain.ChainPolicy.RevocationMode = X509RevocationMode.Online;

chain.ChainPolicy.RevocationFlag = X509RevocationFlag.ExcludeRoot;

chain.ChainPolicy.VerificationFlags = X509VerificationFlags.NoFlag;

chain.ChainPolicy.VerificationTime = DateTime.Now;

chain.ChainPolicy.UrlRetrievalTimeout = new TimeSpan(0, 0, 0);

}

My implementation of *IsTrusted* then checks if either an issuer or a the end certificate itself (specified by the *ValidationMode* property) is in a trust list. The check is done by comparing the thumbprint of the certificate in question against a list.

protected virtual bool IsTrusted(X509Chain chain, string[] trustedThumbprints)

{

int depth = 0;

if (ValidationMode == ValidationMode.EndCertificate)

{

// only check the end certificate

return CheckThumbprint(chain.ChainElements[0].Certificate, trustedThumbprints);

}

else

{

// check the rest of the chain

foreach (X509ChainElement element in chain.ChainElements)

{

if (++depth == 1)

{

continue;

}

if (CheckThumbprint(element.Certificate, trustedThumbprints))

{

return true;

}

}

}

return false;

}

The last step is to register the validator in a *serviceCredentials* behavior.

<serviceCredentials>

<serviceCertificate findValue="Service"

x509FindType="FindBySubjectName"

storeLocation="CurrentUser"

storeName="My" />

<clientCertificate>

<authentication certificateValidationMode="Custom"

customCertificateValidatorType="type" />

</clientCertificate>

</serviceCredentials>

In the download you can find a ready to use validator base class from which you can derive from. You just have to implement the *GetTrustedThumbprints* method and return a *string[]* of thumbprints. You can get the thumbprints from the certificate UI (just remove the blanks). Have fun!