Digitally Signing an XML Document and Verifying the Signature

weblog.west-wind.com/posts/2008/Feb/23/Digitally-Signing-an-XML-Document-and-Verifying-the-Signature

February 23, 2008 - from Maui, Hawaii

Recently I've been involved in some work that deals with signed XML documents outside of a WCF Web service environment. Ironically the service that is being accessed accepts SOAP data but the service does not expose any WSDL even though it uses some WS-* specification features - namely digital signature of the XML body of the message.

Since I didn't have much luck finding the information I needed in one place I thought I'd write it up here, so it hopefully helps out some of you (and myself) in the future.

[Updated: 2/23/08 - added code updates and section for installing live certs]
[Updated: 3/12/08 - added notes about PreserveWhitespace matching]

Certificates for Testing

The most frustrating part of the process for me was getting the certificates set up correctly for testing. Certificate logic - private and public keys - is something that's always been rather fuzzy in my mind and getting the keys set up properly in Windows is really quite a pain in the ass because there are a bunch of different options to do it.

For quick review: When signing a document you'll use a Private key to sign a portion of the the document and embed a signature into the document. You then use a Public key to validate the signature when reading the document. Both signing and and validation can be done with the SignedXml class in combination with the X509Certificate and X509Certifcate2 classes.

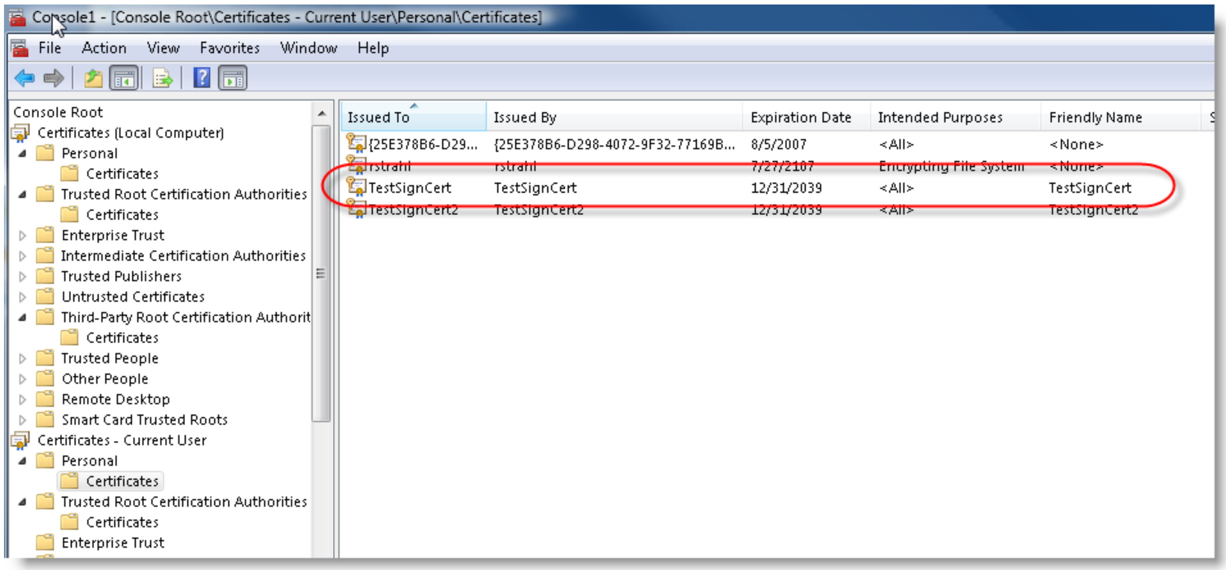
For testing it's a good idea to create self signed certificate that you can use both for signing and validating.

When you create a key with Windows using the *MakeCert.exe* utility you can create a key pair that includes both public and private key and automatically install these keys into the Windows certificate store.

To do this you need to:

- Create the certificate:
  makecert.exe -r -pe -n "CN=TestSignCert" -ss my -sr currentuser -sky exchange -sy 12 "c:\TestSignCert.cer" You can review the options for makecert here. The above creates a key that is self signed (-r), includes a private and exportable key (-pe), the name of the signer (-n in x509 convention), that the location is going to be My or Personal (-ss my) and in the currentuser store (-sr), that the key will be used for message exchange (-sky exchange) and the crypto type as RSA (-sy 12). The public key automatica

- Run **mmc.exe** and **Add the Certificates Add-in**
  Go to the Current User Certificates | Personal and find your certificate (you'll need to use the Local Machine Store later in order to work with IIS so keep that in mind). Make sure the cert **has both the cert and key showing** in the icon (ie. key and cert seal). The key indicates a Private key while the cert seal alone indicates a public key.
- Right click and select Properties and add a Friendly Name (TestSignCert here) - it's blank by default which is not very useful. In code it's easiest to retrieve the friendly name because it's a property of the certificate. All other certificate 'values' (like Issued to or by) are embedded as part of the Subject string and have to be manually parsed out which is more work. Using FriendlyName is easiest.


Console1 - [Console Root\Certificates - Current User\Personal\Certificates]
File
Action
View
Favorites
Window
Help
Console Root
Certificates (Local Computer)
Personal
Certificates
Trusted Root Certification Authorities
Certificates
Enterprise Trust
Intermediate Certification Authorities
Trusted Publishers
Untrusted Certificates
Third-Party Root Certification Authorit
Certificates
Trusted People
Other People
Remote Desktop
Smart Card Trusted Roots
Certificates - Current User
Personal
Certificates
Trusted Root Certification Authorities
Certificates
Enterprise Trust

Issued To	Issued By	Expiration Date	Intended Purposes	Friendly Name
{25E378B6-D29...	{25E378B6-D298-4072-9F32-77169B...	8/5/2007	<All>	<None>
rstrahl	rstrahl	7/27/2107	Encrypting File System	<None>
TestSignCert	TestSignCert	12/31/2039	<All>	TestSignCert
TestSignCert2	TestSignCert2	12/31/2039	<All>	TestSignCert2

The latter step is optional but I prefer to assign a friendly key name so its easier to reference the key. The code below uses Friendly Names to retrieve the key. But you can use any of the other entries. Subject (which is the same value as Issued To) is what key command line tools generally use to look up keys.

At this point you should have a working certificate that you can use for digital signatures.

Signing an XML Document

The next step is to sign an XML document. There are a number of ways to do this and the process will vary based on how your original document is set up. Signing basically involves picking one or more sections in the document that are marked for signing, getting them signed and then embedding the signature into the XML document.

In my scenario I have an XML SOAP Envelope document with just a SOAP:Envelope and SOAP:Body with the body content being the content that is to be signed. The document looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<SOAP:Envelope xmlns:r1="http://www.routeone.com/namespace.messaging.diag#"

               xmlns:star="http://www.starstandards.org/STAR"

               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

               xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"

               xmlns:oa="http://www.openapplications.org/oagis">  <SOAP:Body>

    <!-- data to be signed here --
>                                  </SOAP:Body>  </SOAP:Envelope>
```

Once the document is signed it should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<SOAP:Envelope xmlns:r1="http://www.routeone.com/namespace.messaging.diag#"
xmlns:star="http://www.starstandards.org/STAR" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/" xmlns:oa="http://www.openapplications.org/oagis">

  <SOAP:Header>

    <SOAP-SEC:Signature MustUnderstand="1" xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/2000-
12">


      <Signature xmlns="http://www.w3.org/2000/09/xmldsig#"> <SignedInfo>

          <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
/>

          <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"
/>

          <Reference URI="#Body">            <Transforms>

              <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
/>                                                                                    </Transforms>

            <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"
/>


            <DigestValue>6hpmccmjxQmAI143OhQfIWpkryw=</DigestValue> </Reference>
```

```
</SignedInfo>

<SignatureValue>sv8n4h0rV4Xmbl+M+w+MLl7lVA8KFsoWRx5DqSKkwSie32jOFoJt0WvH6UWRQInW5XpAL3OtNZcw8pbCdTtl8KSo7UIkl1

<KeyInfo>                   <X509Data>            <X509IssuerSerial>

<X509IssuerName>CN=TestSignCert</X509IssuerName>

<X509SerialNumber>75496503122422458150193540449068096025</X509SerialNumber>

</X509IssuerSerial>            </X509Data>          </KeyInfo>        </Signature>

    </SOAP-                                                           <!-- data to be signed here --
SEC:Signature>           </SOAP:Header>    <SOAP:Body id="Body"> >

</SOAP:Body>  </SOAP:Envelope>
```

Note that the SOAP:Body tag gets an id attribute appended - the id is referenced with <Reference URI="#Body"> in the signature section. This points at the section (or more than one) that is to be signed.

The code to encode the document is shown below. Please note that the code is somewhat generic and should work with most SOAP envelope documents:

```
              /// Signs the SOAP document and adds a digital signature to
/// <summary> it.                                                        ///

/// Note a lot of optional settings are applied
against

/// key and certificate info to match the required XML         /// structure the server
document                                                       requests.

/// </summary> /// <param name="xmlDoc"></param>

/// <param name="certFriendlyName">Friendly Name of Cert installed in the Certificate Store under
CurrentUser | Personal</param>

                        public XmlDocument SignSoapBody(XmlDocument xmlDoc, X509Certificate2
/// <returns></returns> cert)                                                                   {

    // *** Add search Namespaces references to ensure we can reliably work

    // *** against any SOAP docs regardless of tag
naming

    XmlNamespaceManager ns = new
XmlNamespaceManager(xmlDoc.NameTable);

    ns.AddNamespace("SOAP",
STR_SOAP_NS);                             ns.AddNamespace("SOAP-SEC", STR_SOAPSEC_NS);
```

```
    // *** Grab the body element - this is what we create the signature
from

    XmlElement body = xmlDoc.DocumentElement.SelectSingleNode(@"//SOAP:Body", ns) as
XmlElement;

    if (body ==              throw new ApplicationException("No body tag
null)                  found");

    // *** We'll only encode the <SOAP:Body> - add id: Reference as
#Body

    body.SetAttribute("id",              // *** Signed XML will create Xml Signature - Xml
"Body");                            fragment

    SignedXml signedXml = new                          // *** Create a KeyInfo
SignedXml(xmlDoc);                                   structure

    KeyInfo keyInfo = new                // *** The actual key for signing - MAKE SURE THIS ISN'T
KeyInfo();                          NULL!

    signedXml.SigningKey =
cert.PrivateKey;

    // *** Specifically use the issuer and serial number for the data rather than the
default

    KeyInfoX509Data keyInfoData = new
KeyInfoX509Data();

    keyInfoData.AddIssuerSerial(cert.Issuer, cert.GetSerialNumberString());

    keyInfo.AddClause(keyInfoData);

    // *** provide the certficate info that gets embedded - note this is
only

    // *** for specific formatting of the message to provide the cert          signedXml.KeyInfo =
info                                                                      keyInfo;

       // *** Again unusual - meant to make the document match
   template

    signedXml.SignedInfo.CanonicalizationMethod =
SignedXml.XmlDsigExcC14NTransformUrl;

    // *** Now create reference to sign: Point at the Body
element

    Reference reference = new
Reference();

    reference.Uri = "#Body";  // reference id=body section in same
doc

    reference.AddTransform(new XmlDsigExcC14NTransform());  // required to match
doc

                                                // *** Finally create the
signedXml.AddReference(reference);      signature
```

```
                                         // *** Result is an XML node with the signature detail below
signedXml.ComputeSignature();      it


    // *** Now let's add the sucker into the SOAP-           XmlElement signedElement =
HEADER                                                   signedXml.GetXml();


      // *** Create SOAP-SEC:Signature
 element


    XmlElement soapSignature = xmlDoc.CreateElement("Signature",
STR_SOAPSEC_NS);


    soapSignature.Prefix = "SOAP-
SEC";                                    soapSignature.SetAttribute("MustUnderstand", "", "1");


    // *** And add our signature as
content                                       soapSignature.AppendChild(signedElement);


    // *** Now add the signature header into the master
header


    XmlElement soapHeader = xmlDoc.DocumentElement.SelectSingleNode("//SOAP:Header", ns) as
XmlElement;


    if (soapHeader ==                    soapHeader = xmlDoc.CreateElement("Header",
null)                          { STR_SOAP_NS);


        soapHeader.Prefix =
"SOAP";


        xmlDoc.DocumentElement.InsertBefore(soapHeader,
xmlDoc.DocumentElement.ChildNodes[0]);                                                     }


                                              return
    soapHeader.AppendChild(soapSignature);  xmlDoc;           }
```

The signing code is a little more complex than it needs to be, mainly because I had to match the signature settings exactly to what the server expects. In simpler scenarios you may not have specify things like CanonicalizationMethod or the security provider or transform - especially if you're using .NET on both ends of the connection.

Note also that to write you're using the Private key so the key is being read out of the Certificate store. You can also read a certificate from a file, but this is not recommended given that the private key should be private and safely stored away. The easiest way to keep it isolated is in the certificate store. .NET 3.0 makes this very easy with the various X509 related certificate classes which are pretty easy to use.

To then validate the signature on the resulting XmlDocument you can use code like the following:

```
             /// Validates the Xml Signature in a
/// <summary> document.                                      ///


/// This routine is significantly simpler because the key
parameters


/// are embedded into the signature itself. All that's needed is
a


/// certificate to provide the key - the rest can be read from     /// Signature
the                                                                itself.              /// </summary>


/// <param name="doc"></param> /// <param name="publicCertFileName"></param> /// <returns></returns>


public bool ValidateSoapBodySignature(XmlDocument doc, X509Certificate2 cert) {
    // *** Load the doc this          SignedXml sdoc = new
time                              SignedXml(doc);

    // *** Find the signature and load it into
SignedXml

    XmlNodeList nodeList =
doc.GetElementsByTagName("Signature");

                                                // *** Now read the actual signature and
    sdoc.LoadXml((XmlElement)nodeList[0]);  validate

    bool result = sdoc.CheckSignature(cert,             return
true);                                             result;           }
```

The validation process is a heck of a lot easier because all of the information regarding keys and certificate information is all provided as part of the signature that's embedded in the document. You simply provide the cert and call CheckSignature() and off you go. Compared to signing this is a walk in the park.

Getting the Certificate

One thing that the code above doesn't really show is retrieving the certificate from the Certificate Store. The following methods (on the same class above) demonstrate how to retrieve a certificate from the store by its 'subject name' (ie. Issued To) value and from a file.

```
             /// Retrieve a Certificate from the Windows Certificate      /// by its Friendly
/// <summary> store                                                      name.

              /// <param name="subject">The friendly name of the
/// </summary> certificate</param>

/// <param name="storeName">The store name type ( for example: Storename.My
)</param>

/// <param name="storeLocation">Top level Location (CurrentUser,LocalMachine)</param>

/// <returns></returns>

public X509Certificate2 GetCertificateBySubject(string subject, StoreName storeName, StoreLocation
storeLocation)

     X509Store xstore = new X509Store(storeName,
{ storeLocation);

    xstore.Open(OpenFlags.ReadOnly |                                   X509Certificate2 cert =
OpenFlags.OpenExistingOnly);                                      null;

    foreach (X509Certificate2 cert2 in xstore.Certificates)     {

        string sub = wwUtils.ExtractString(cert2.Subject, "CN=", ",", true,
true);

        if (subject ==                          cert =
sub)                         {        cert2;                              break; }            }

                                 /// Retrieve a Certificate from the Windows Certificate
    return cert; }  /// <summary> store

/// by its Friendly           /// This code pulls from CurrentUser - Personal cert
name.                     ///  store                                                  /// </summary>

/// <param name="subject"></param> /// <returns></returns>
```

```
public X509Certificate2 GetCertificateBySubject(string subject) {

    return this.GetCertificateBySubject(subject, StoreName.My, StoreLocation.CurrentUser); }  /// <summary>

/// Creates a Certificate from a
file                                  /// </summary> /// <param name="fileName"></param>


                        public X509Certificate2 GetCertificateFromFile(string
/// <returns></returns> fileName)                                                       {

    X509Certificate cert =
X509Certificate.CreateFromCertFile(fileName);

    return new X509Certificate2(cert); }
```

As I mentioned earlier the Subject (or Issued To) value requires parsing and I use a library routine to extract a string the string. If you can ensure you have a friendly name on the key you can directly compare the FriendlyName property of the cert instead. The last method is there just for completelness - it doesn't add much value since it simply reads the cert from a file.

Watch out for PreserveWhiteSpace

One thing you'll want to be very careful of when signing documents is the PreserveWhiteSpace property on the XmlDocument object. This property must be set the same way for signing and validation. By default .NET has PreserveWhiteSpace off and in my experience most other solutions that use signatures expect PreserveWhiteSpace on. If you mismatch the white space preservation between encoding and validation, the validation will fail even if everything else is correct. You'd figure that the PreserveWhiteSpace setting would be part of the signature, but this must be done explicitly.

In your .NET code this basically means that when you load your XmlDocument you should immediately set the PreserveWhitespace property (if you need it set to true that is). This ensures that your encoding will respect the setting and also ensures documents get saved properly if you write them to disk. Make sure to set the property BEFORE loading any content into XmlDocument with Load() or LoadXml(). If your application will need to use the setting (or any other settings on XmlDocument) consistently you should create a factory method and always use it to create instances. For example on our processor object these three methods exist to load a document:

```
/// Gets a properly formatted instance of an Xml
document                                                  /// </summary> /// <returns></returns>


                                         XmlDocument doc = new
public XmlDocument GetXmlDocument() { XmlDocument();

    doc.PreserveWhitespace =
true;                                   return doc; }  /// <summary>

/// Gets a properly formatted Xml Document from an input
stream                                                          /// </summary>


                                                         public XmlDocument GetXmlDocument(Stream
/// <param name="stream"></param> /// <returns></returns> stream)                                         {

    XmlDocument doc =
this.GetXmlDocument();                          doc.Load(stream);     return doc; }  /// <summary>

/// Gets a properly formatted Xml document from a file
name                                                        /// </summary>

/// <param name="fileName"></param> /// <returns></returns>

public XmlDocument GetXmlDocument(string fileName) {

    XmlDocument doc = this.GetXmlDocument();                doc.Load(fileName);     return doc; }
```

Installing Actual Keys from Thawte

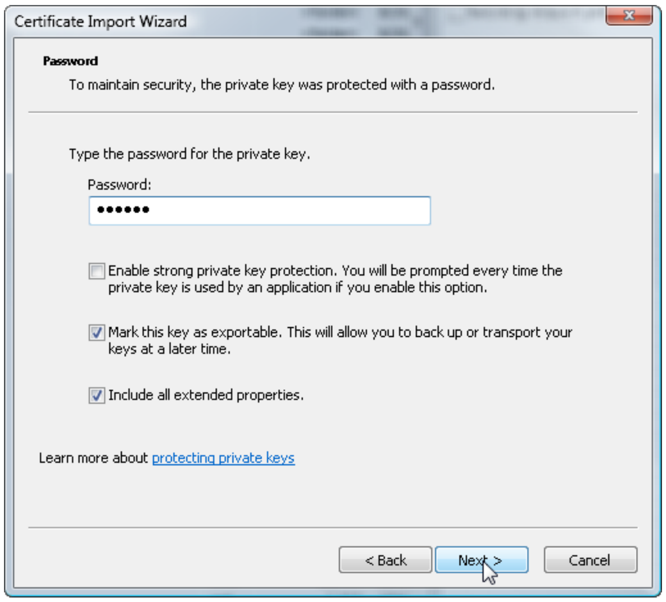
Once your testing is all done you'll probably want to install a real key from a certifcate authority. In my case we were using Thawte as our CA to provide the key.

When you receive your final keys from the Thawte you end up with a .PVK and .SPC file. The .pvk is the private key that was generated as part of the certificate request. The .spc contains the merged key files. As it is these two files aren't useful until you merge them and import them into the Windows Certificate Store.

You'll need to use a special tool to do this: PVK2PFX.EXE from the Windows SDK (my version is 6.0). The SDK installs with Visual Studio (if you didn't diable the option) and you can find the file in C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin.

pvk2pfx.exe -pvk Westwind.pvk -spc Westwind.spc -pfx Westwind.pfx

This will popup and prompt for your private key password and give you options on how to import the key into the certificate store.


Certificate Import Wizard

Password

To maintain security, the private key was protected with a password.

Type the password for the private key.

Password: ••••••

[ ] Enable strong private key protection. You will be prompted every time the private key is used by an application if you enable this option.

[✓] Mark this key as exportable. This will allow you to back up or transport your keys at a later time.

[✓] Include all extended properties.

Learn more about protecting private keys

Make sure that you mark the key as exportable which ensures that you can more easily move the key around without importing it first again. Don't check the first option - if you do a Windows dialog will pop up every time you try to use the private key even if you access it programmatically.

The next form prompts you were to store the key. If you accept automatic installation the key is stored in your CurentUser Personal store (StoreName.My, StoreLocation.CurrentUser). Alternately you can select the store to store your key in, but keep in mind that the store selection is limited to the current user.

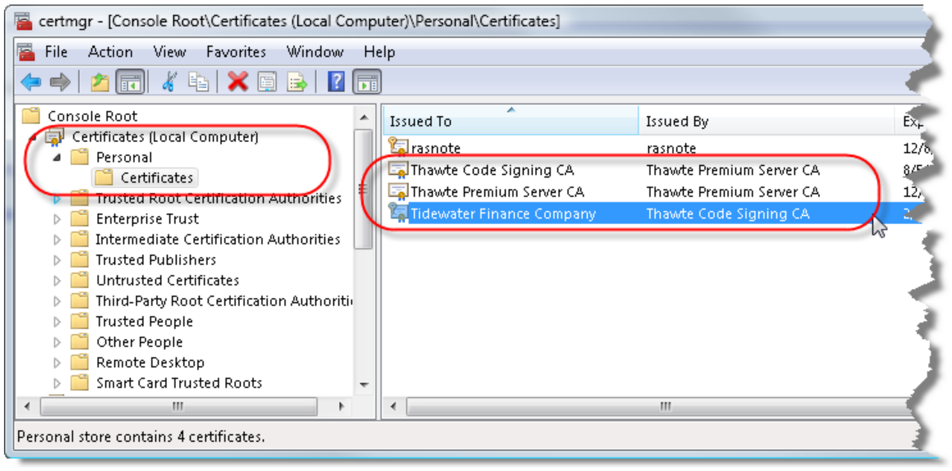
When you're done you end up with .PFX file that is ready to be installed on other machines and the key installed in the current user Personal store.

IIS and Certificates

If you plan on using your key from a Web application, you'll have to copy the key from your Current User Store to the LocalMachine and then use StoreLocation.LocalMachine when specifying the store to look for the key.

The most reliable way to get the key installed into the Local Machine Certificate store. Open the certificate console and add the Local Machine Certificate snap-in. When you're there go to the personal location and then select Import Certificates from the context menu and import the key from the PFX you generated in the last step.

You should end with something like this:


certmgr - [Console Root\Certificates (Local Computer)\Personal\Certificates]
File Action View Favorites Window Help
Console Root
Certificates (Local Computer)
Personal
Certificates
Trusted Root Certification Authorities
Enterprise Trust
Intermediate Certification Authorities
Trusted Publishers
Untrusted Certificates
Third-Party Root Certification Authoriti
Trusted People
Other People
Remote Desktop
Smart Card Trusted Roots

Issued To	Issued By	Ex
rasnote	rasnote	12/
Thawte Code Signing CA	Thawte Premium Server CA	8/
Thawte Premium Server CA	Thawte Premium Server CA	12/
Tidewater Finance Company	Thawte Code Signing CA	2

Personal store contains 4 certificates.

This will now let you access the store using StoreName.My and StoreLocation.LocalMachine which means this key is globally accessible. Remember to set the Friendly Name after import so the code above can find the key.

IIS Permission Requirements - Adding NETWORK SERVICE

But you're not quite done yet if you need to access the key with IIS. I spent about an hour trying to figure out why the the Private Key kept failing under IIS but was working fine in my tests. It turns out the problem is that IIS (ie. NETWORK SERVICE) doesn't have rights to read the private key data from the store and so although the certificate loads, the private key fails when assigned.

Unfortunately unless you're running your Web application under SYSTEM or an Admin you will need to perform yet another configuration step to allow access to the key information to the Web user account. The process for this is outlined in a blog post here and it's mighty ugly. It essentially boils down to setting permissions on the Windows internal key files either manually or using a tool. Many thanks to Christian Weyer, who when pinged spotted the problem immediately.

In a nutshell you need to download WinHttpCertCfg.exe (download) and then add permissions to your installed key in the Certficate Store. The command line to do so is:

C:\>winhttpcertcfg -c LOCAL_MACHINE\my -s "West Wind Code Signing" -g -a NETWORK_SERVICE

-g is grant and -a is account, -s the subject and -c the store.The subject is the same as the Issued to value in the Console view.

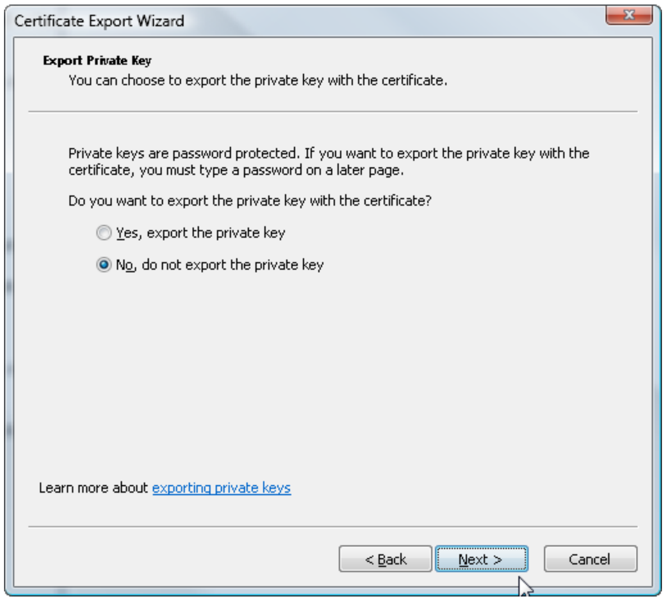
And that should hopefully do the trick getting your key registered and usable with IIS. I say hopefully because unless you follow the steps exactly there's a good chance that something can go wrong as it did for me, mainly because I didn't find this information all in one place, but rather scattered over about 20 different sources.

Exporting the Public Key to .CER File

Once you have the PFX file, you can use the same procedure to install the key on other machines as needed. Remember that the key contains both the public and private keys, so you'll want to minimize exposure of this key and limit it to machines that really need to use it - you don't want to compromise the private key in sensitive applications.

Once the key is working for you locally you should probably create a .CER file of the public key that you can send to folks that need to validate the content you are signing or encrypting - in my case the host service provider. To export the key select it in the certificate manager and select All Tasks | Export.

Here's what you should see:


Certificate Export Wizard

Export Private Key
You can choose to export the private key with the certificate.

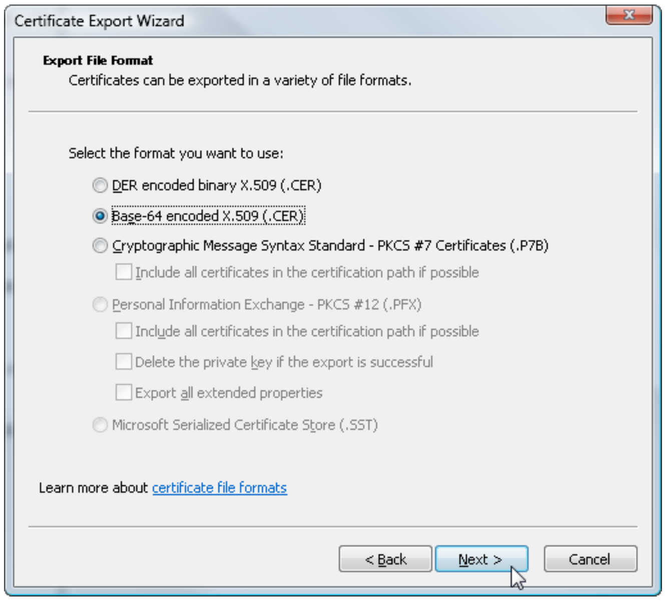
Private keys are password protected. If you want to export the private key with the certificate, you must type a password on a later page.

Do you want to export the private key with the certificate?

Yes, export the private key [✓ No, do not export the private key]

Learn more about exporting private keys

Don't export the private key (even if the option is available). You'll want to create a base 64 .CER file since that's the most widely understood certificate format:


Certificate Export Wizard

Export File Format
Certificates can be exported in a variety of file formats.

Select the format you want to use:

DER encoded binary X.509 (.CER)
[✓ Base-64 encoded X.509 (.CER)]
Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)
Include all certificates in the certification path if possible
Personal Information Exchange - PKCS #12 (.PFX)
Include all certificates in the certification path if possible
Delete the private key if the export is successful
Export all extended properties
Microsoft Serialized Certificate Store (.SST)

Learn more about certificate file formats

Pick a filename and you've got your public key that you can send along with signed messages.

Once you've exported the public key to a .cer file you should be able to sign the document from the certificate store and validate either with the the same certificate certificate from the certificate store (since it contains both the private and public keys) or the .cer file just exported.

There you have it - full round tripping of digital signatures. Not a trivial task to say the least. Hopefully having this information all in one place will be useful to some of you.

A few thoughts

<rant>

Man, was this a piece of freaking work! It really is no wonder that security for signatures and encryption with certificates is so lightly used on Windows. It's a royal pain in the ass, horribly documented and requires a host of tools. Why isn't this stuff all provided through the Management Console Snap In? Adding permissions would certainly be nice...

WinHttpCertCfg looks promising to do everything from importing to the appropriate store, and assigning rights but unfortunately I couldn't get it to work to import my PFX keys into the Local_Machine store. No error, no message it just didn't work...

The following should have worked but didn't:

```
C:\>winhttpcertcfg -c LOCAL_MACHINE\My  -a NETWORK_SERVICE -i WestWindCodeSigning.pfx
```

No error, but also no imported key which is bogus to the max. At least tell me something - what failed but no it just gives a blank report. I searched both Local_Machine and Current_User stores but no luck - they weren't imported.

It really bugs me that these tools are scattered all over the place and not available on the system in the first place and have to be downloaded and then worst of all seem to fail to work. Talk about non-discoverable and obtuse.

This is security by worthless and most likely unintended obfuscation... and in this scenario I'd argue it definitely hurts adoption of good practices in terms of security.

</rant>