Securing kbmMW with StreamSecII SSL

for kbmMW v. 1.00+

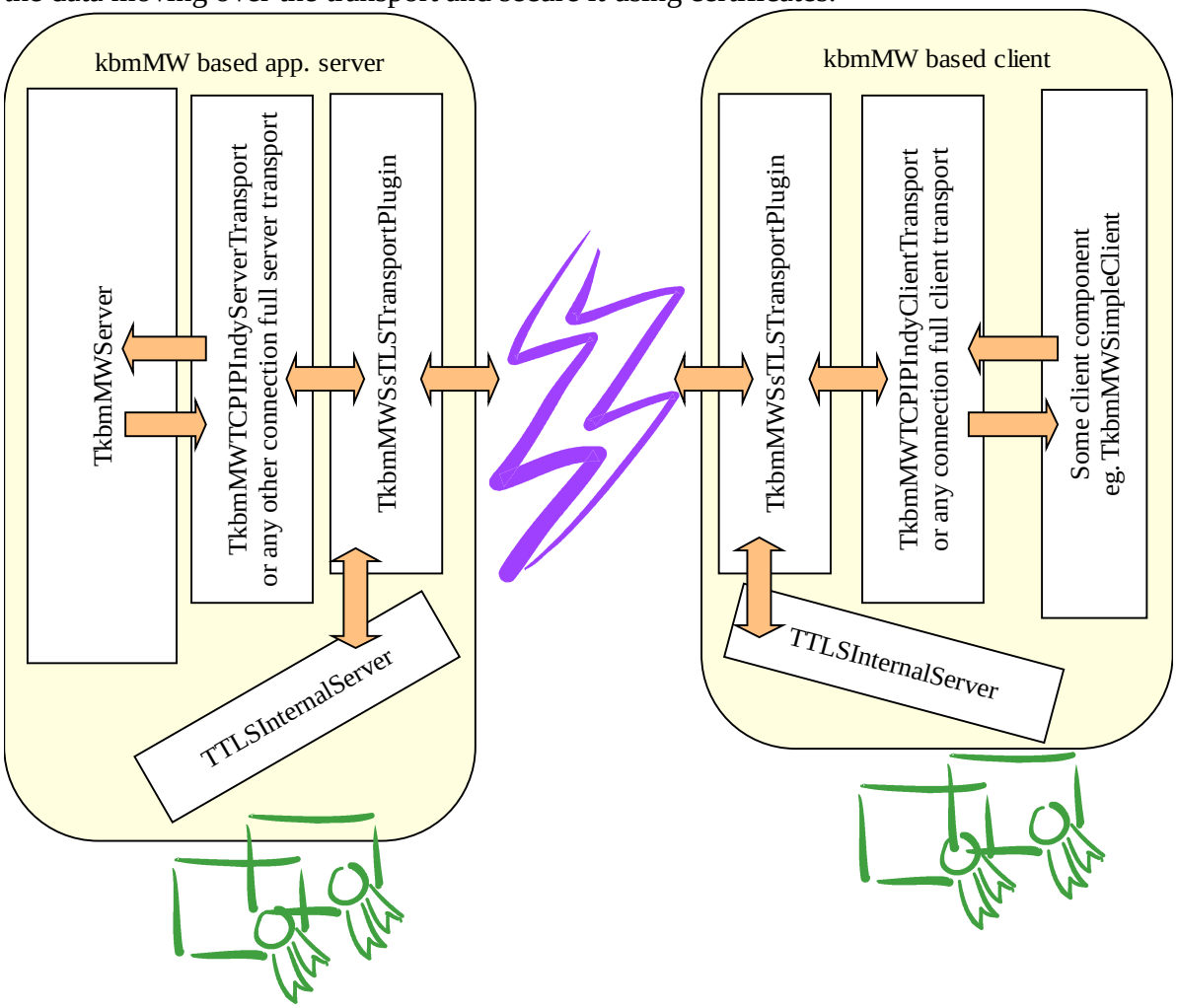
Intro

When you have a client and application server that you know works, then you can add enhanced cryptographic support in the form of SSL. This document describes how to add basic SSL support. It does however not describe all the details of the possible SSL and certificate settings. For this, please consult StreamSec.

Note: The parts of this document that relate to certificate creation does not apply to Linux, but only to Win32.

Basic layout

Adding SSL support to a kbmMW based server or client is really very easy. Simply add a special transport plug-in and connect it to the transport. The plug-in allows for the SSL process to intercept the data moving over the transport and secure it using certificates.



What is SSL ?

SSL stands for Secure Sockets Layer and was originally designed by Netscape Corp. The Internet Engineering Task Force (IETF) now issues all new revisions of SSL. IETF changed the name from SSL to TLS, which stands for Transport Layer Security. SSL 3.0 was the last version issued by Netscape, and TLS 1.0 can be thought of as “SSL 3.1”.

SSL is a method for securing the information a client and a server send to each other. When you move data over the Internet or for that matter even a local area network, you have very little privacy. People can monitor, manipulate and forge what you reveal to, or request from, the server. Credit card numbers, personal data, or controversial information is an open book to the technologically sophisticated attacker.

- SSL was designed to defeat the snoops and protect your privacy. SSL guarantees the *confidentiality* of your data.
- SSL will detect any manipulation of the data while it travels through the network. SSL guarantees the *integrity* of your data.
- SSL will stop impostors at the door. You will be told who you connected to and no one else will be capable of stepping in and start sending data. SSL guarantees that the data you receive is *authentic*.

An SSL-enhanced client use encryption to scramble the data you send to an application server into an unintelligible string of seemingly random characters.

A transaction could for example be a client sending a request containing a credit card number to the application server.

Let's look at an example showing the difference between unsecure and secure transactions:

1. Unsecure transaction:
The client will send the credit card number in plain text "3333-4444-5555-6666" to the server.

2. Secure (SSL) Transaction:
The client converts "3333-4444-5555-6666" into a seemingly random collection of characters like "e$$%0lj*&*(#foij" and sends it to the server. The server receives "e$$%0lj*&*(#foij" and converts it back into "3333-4444-5555-6666".

The important thing to notice here is that when the client encrypts the data no one can read the contents and obtain your personal details.

SSL will also employ even more advanced codes to guarantee the integrity of the data. Suppose you send the following message:

“Pay US$ 00000199 to StreamSec HB”

Even if this message is encrypted before it leaves your computer, a cunning attacker might be capable of manipulating the data in ways you don't want. Suppose some of the 9th to 16th characters are manipulated and the string decrypts to:

"Pay US$ 93265776 to StreamSec HB"

This is what integrity control is supposed to prevent.

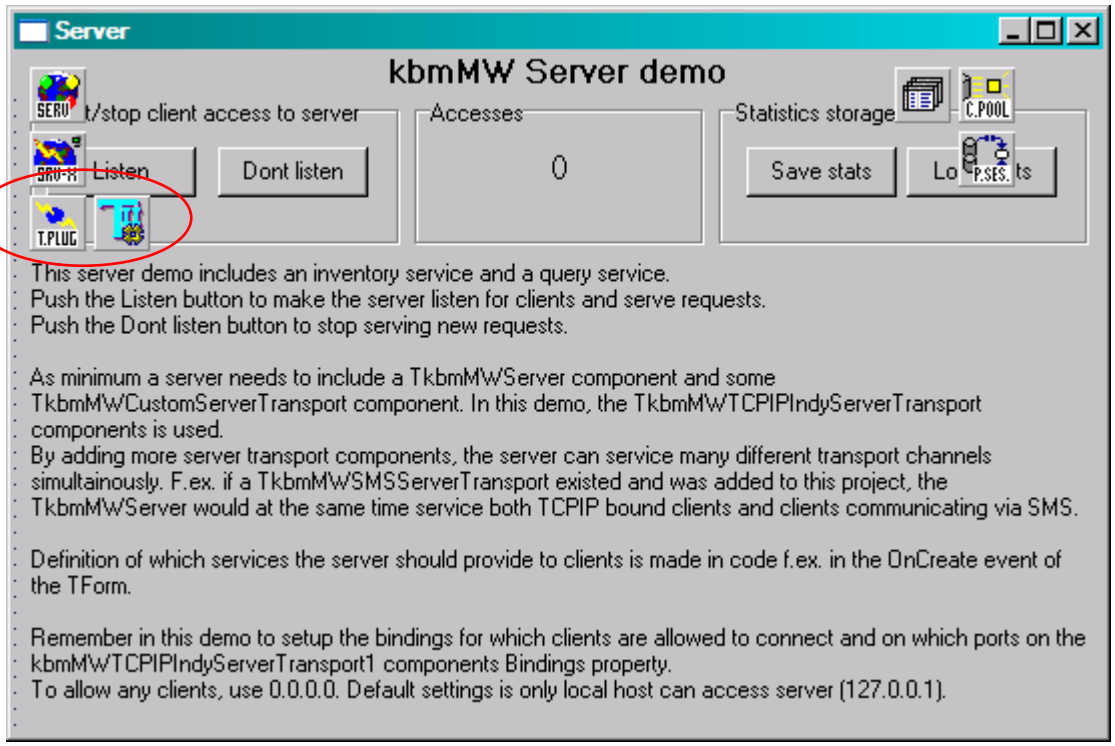
In addition to providing confidentiality and integrity, SSL was designed to answer a related question: How do you know you are really communicating with the application server you intended, and how can the application server know it's really communicating with an approved client? Because SSL operates with asymmetric encryption – that means that the application server have a key which only the application server knows about and a public key which all clients must know if they want to communicate with the server, and the same with the client holding a private key and giving a public key to the application server – both ends can rely on that the other end is indeed the client or server they expect it to be, and not a fraud.

These public keys are passed between the application server and client as digital signature certificates.

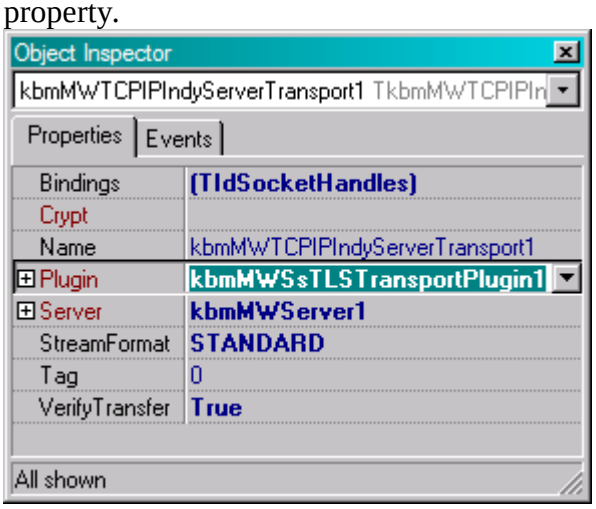
A digital signature certificate is an encrypted electronic document that contains information that verifies your company's identity, encoded in a highly secure format that makes the information impossible to forge. Your server will send your digital signature certificate and electronic counter signature to your customers to authenticate your identity to them, and your customer's browser software sends you their digital signature certificate and electronic counter signature containing verification of their identity.

How to secure an existing application server?

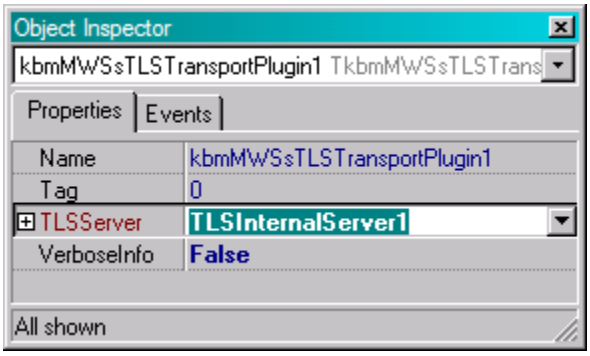
If you already have a working application server that you want to secure using SSL you need to add a TkbmMWSsTransportPlugin and a TSimpleTLSInternalServer component to the main form/datamodule holding the TkbmMWServer component.



Then link the TkbmMWSsTLSTransportPlugin to the TkbmMWTCPIPServerTransport's Plugin property.



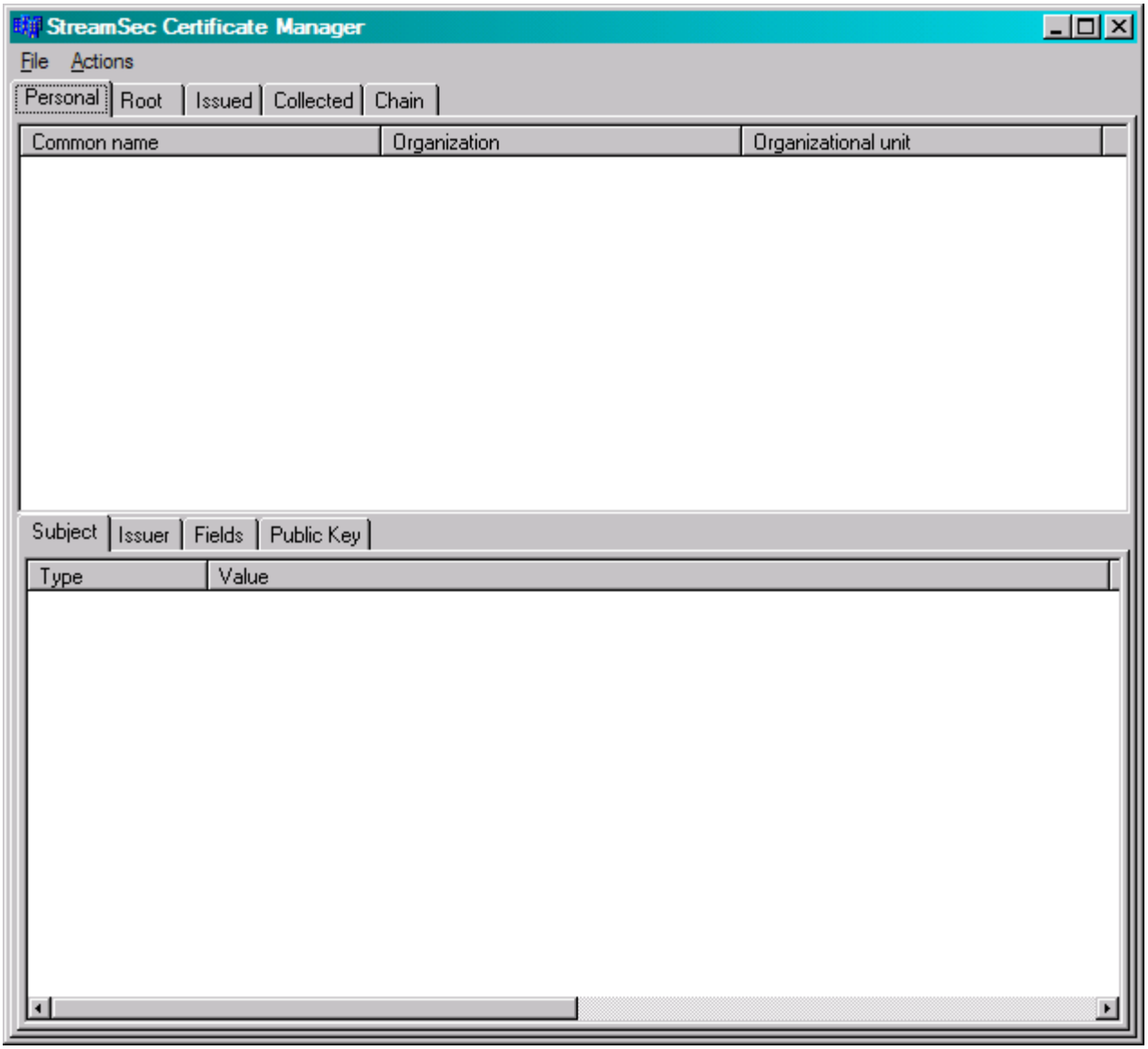
And the TTLSInternalServer to the TkbmMWSsTransportPlugin's TLSServer property.



That is really all what's needed to change in the application server as such.
Now all what's missing is configuration of SSL certificates and generation of public and private keys and key rings.

This document will continue to explain step by step what to do, but will not explain all the SSL and certificate related theories and features. Look in the end of this document for links to documents explaining those subjects in more detail.

For certificate management you will start to use StreamSec Certificate Manager (certmgr.exe), which is part of your StreamSecII installation. It includes full source, but the exe file is also provided. Just use it.

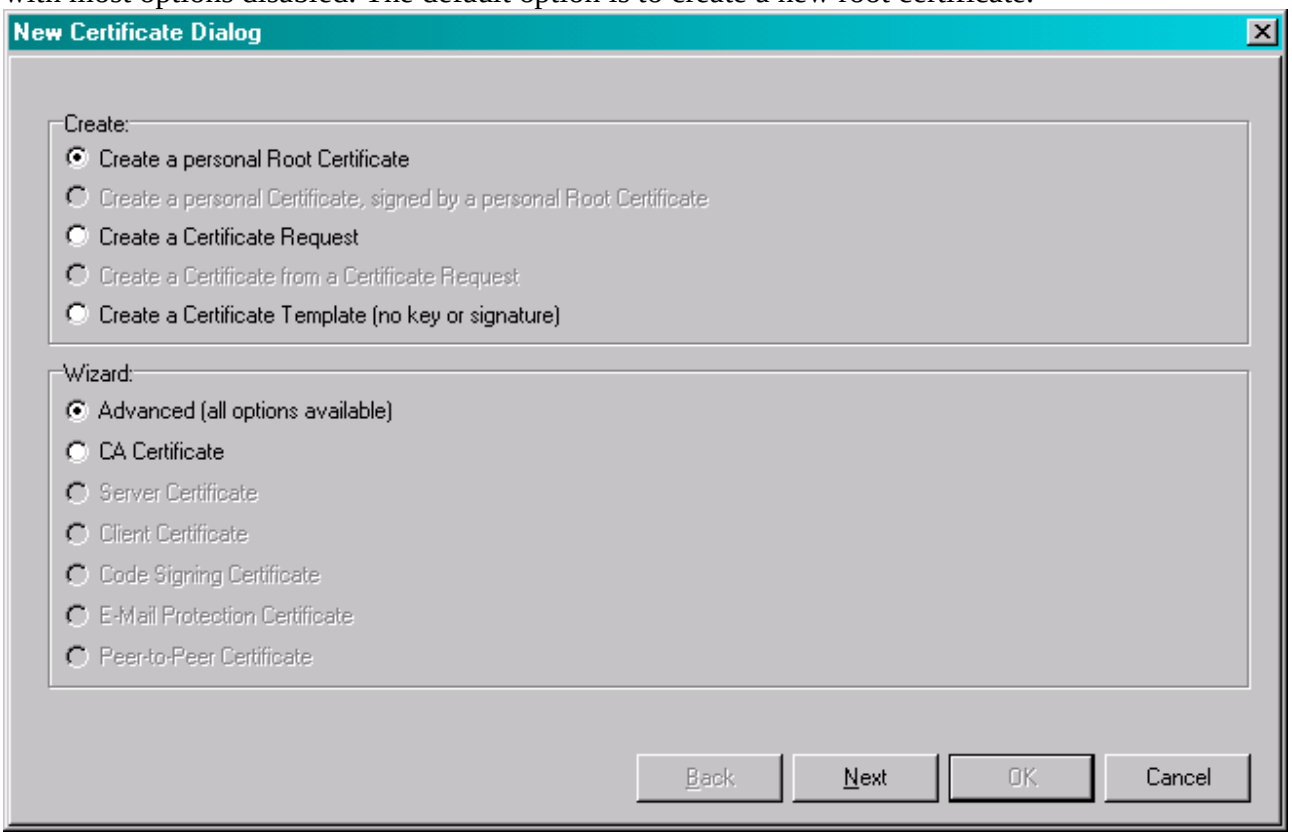


If you don't already have a root certificate (which is also named a CA certificate – Certificate Authority), you need to create one. The root certificate is a digital document describing you or your company as a top-level entity. The root certificate will automatically be used to sign all other certificates you will create later. Well-known companies that specialize in issuing certificates to users also have a root certificate. It is used for verifying who have issued a server or client certificate and this way to know if a trusted party issued the server or client certificate. By creating your own root certificate you technically become your own Verisign.

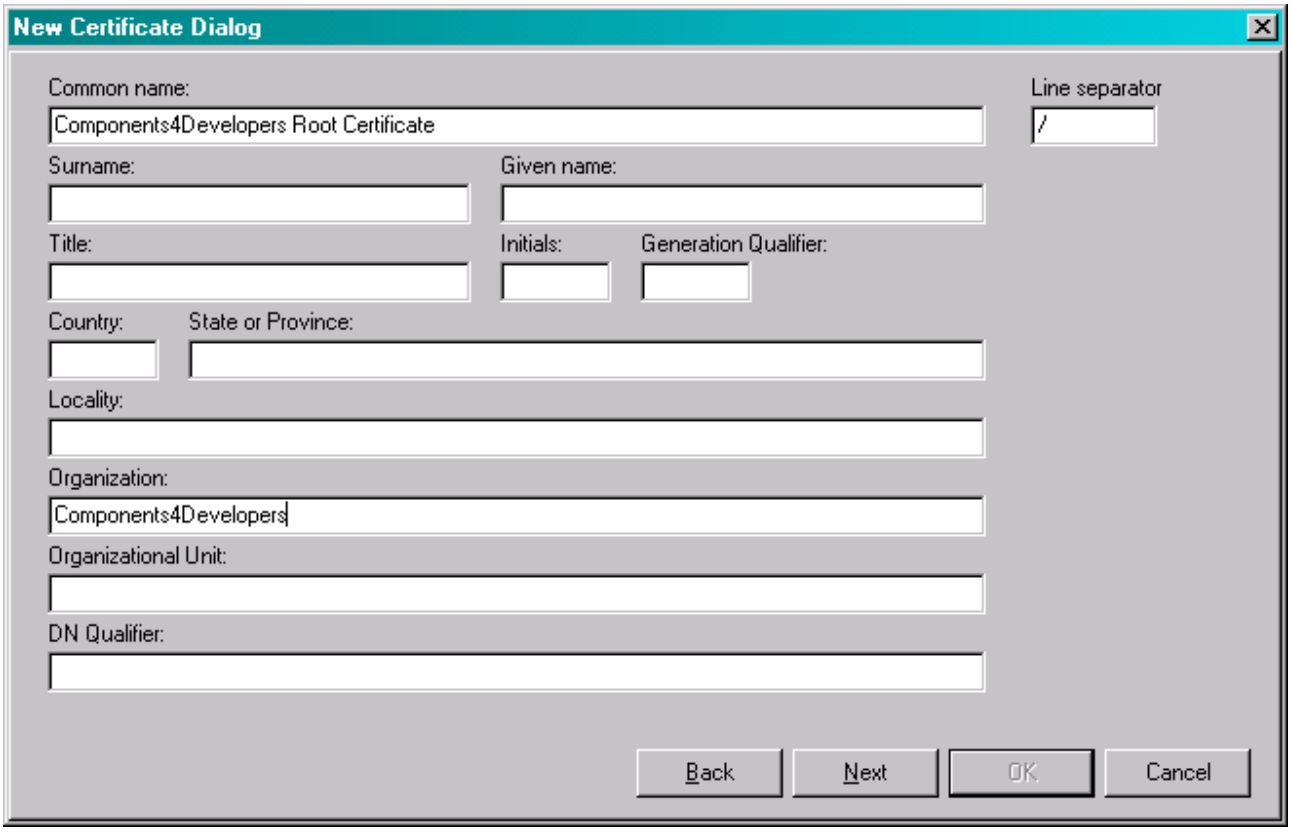
As with all certificates they consist of some information about the certificate (the so-called subject), a private key that you must guard with your life, and a public key that can be delivered to other parties.

So let's start creating the root certificate:

Select Actions|New certificate.
Since this is first time creating certificates, and no root certificate is defined, a dialog will show with most options disabled. The default option is to create a new root certificate.



Select Next.

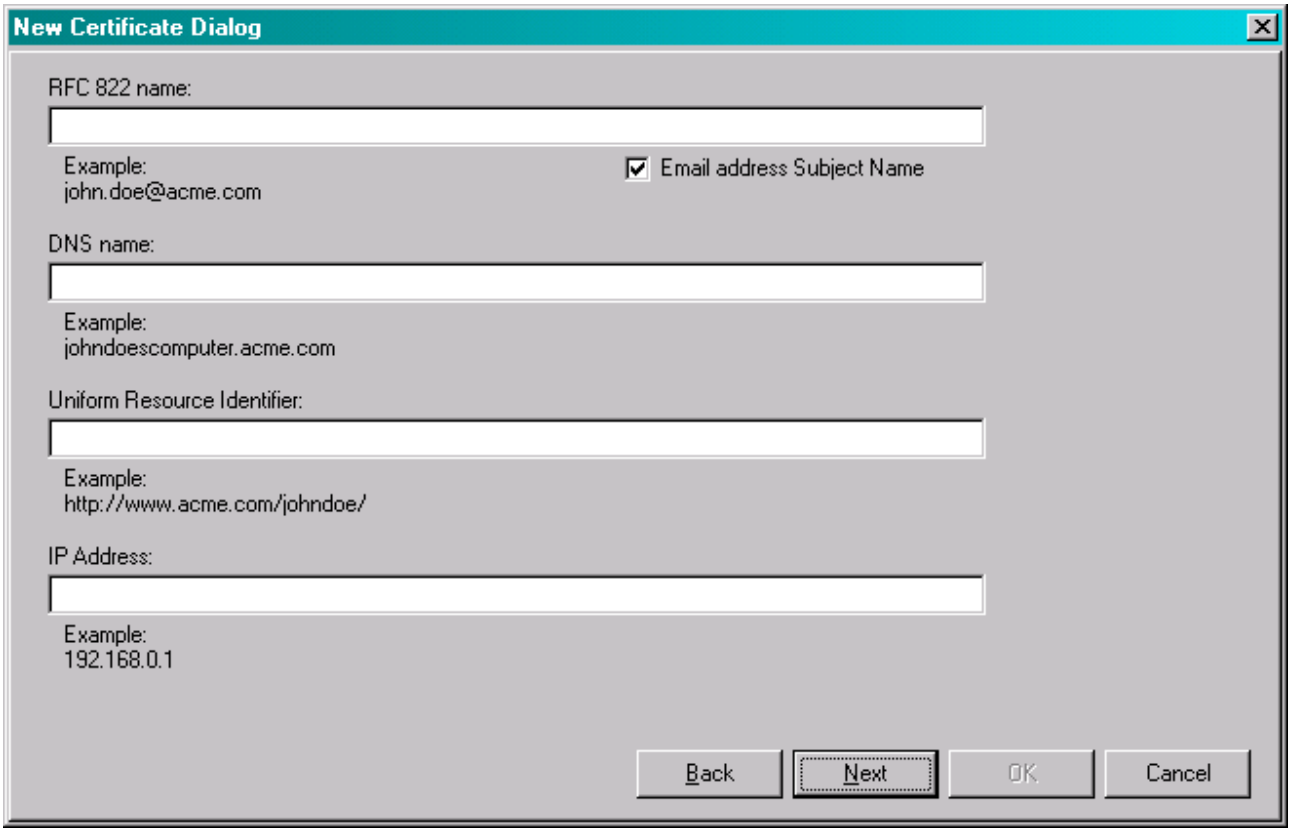


Fill out as many subject fields as you like. You may want to fill out many of them to uniquely identify your company or your self, depending on who is the owner of the root certificate.

Never create two root certificates containing exactly the same subject info. If necessary, use the DN Qualifier field to distinguish between subject names that would otherwise be identical.

Click Next.

This page gives access to enter so called extended subject information. Since you are creating a root certificate, you will not need to fill this one out.

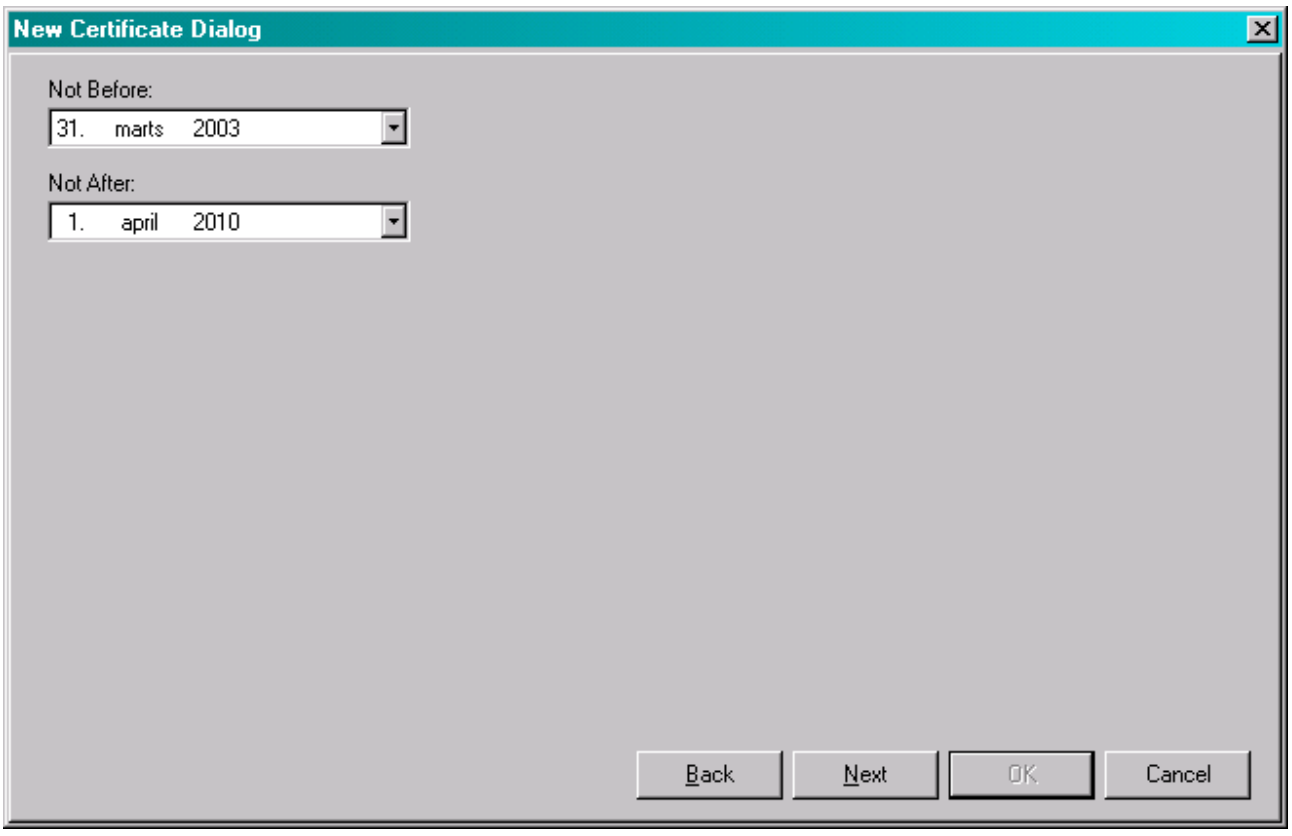


Click Next.

All certificates must have a validity period. Outside that period, the certificate is no longer valid, and any software relying on the certificate will cease to function.
The length of the period is a trade-off between security and practicality.
The longer the period, the less times you will have to issue new updated certificates to users and servers. But if your certificate for some reason is compromised (someone stealing the private key or whatever) you won't be able to easily stop people misusing it within a reasonable timeframe.

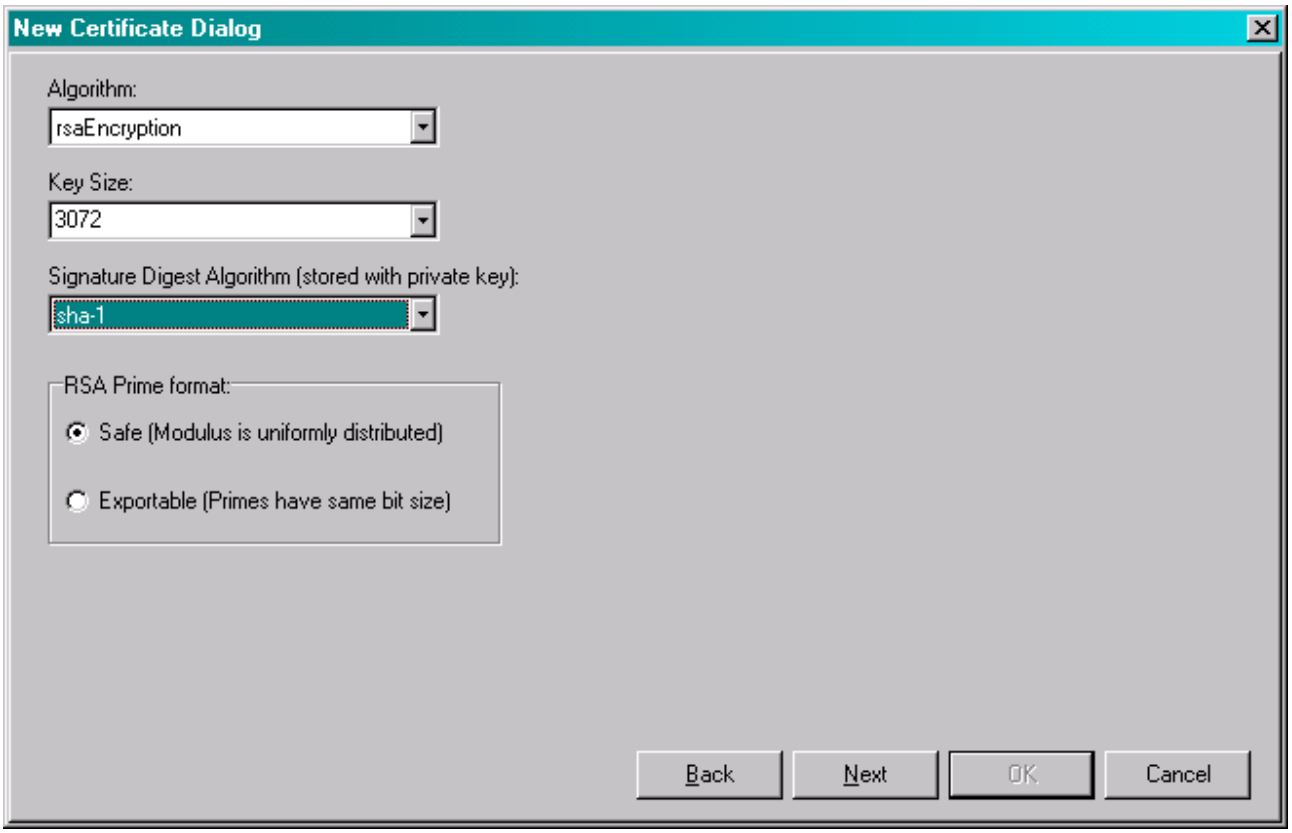
If you make sure to keep your root certificate in a steel enforced concrete vault 200 meters under ground with lots of extremely loyal guards guarding it, you can select a long period like is shown here.

You can however choose to let certificates generated with this root certificate be revocable. See more later on.



Press Next.

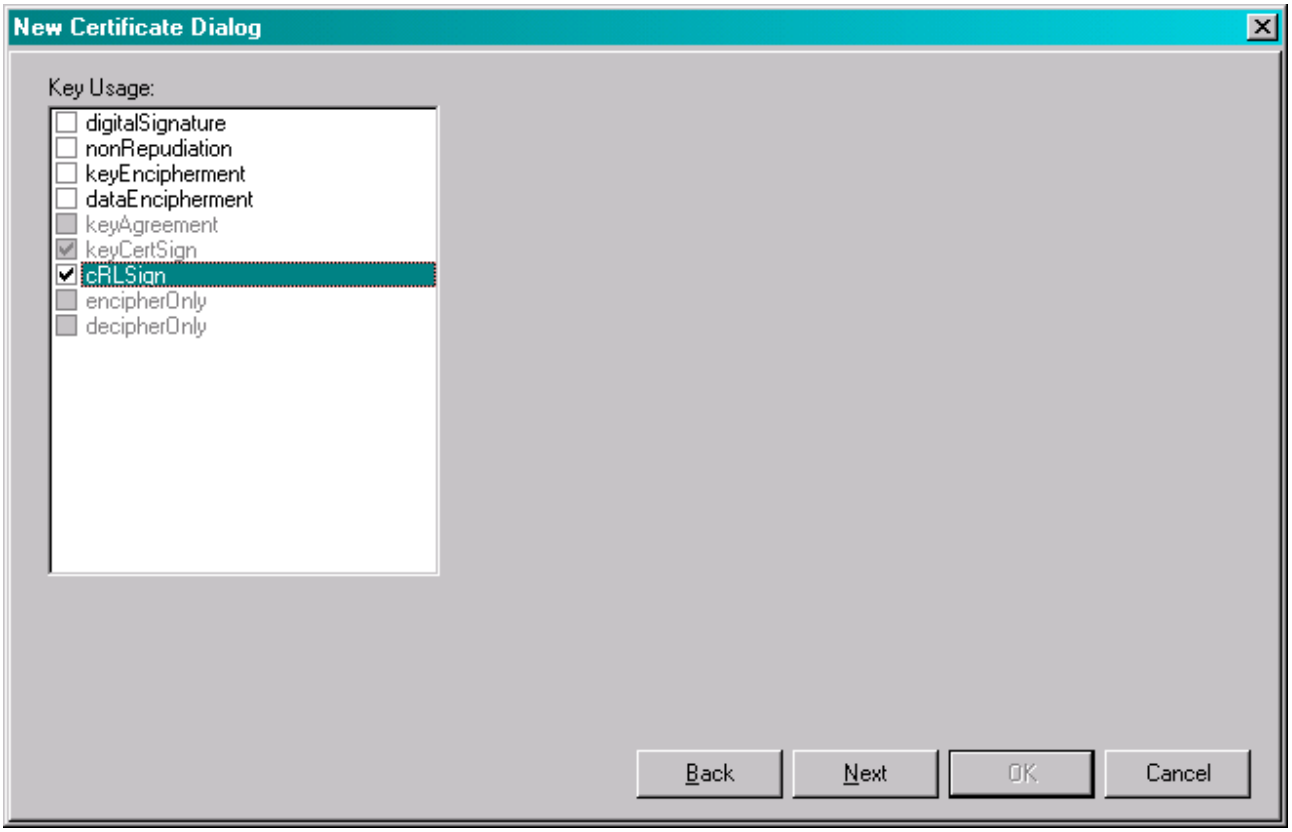
Now you need to select what type of encryption your keys should use. The standard is to use RSA and SHA-1 as selected here.
If you know what you are doing you can choose other settings. But beware that the RSA encryption is no longer under any patents, while the other encryption methods may be partly covered in patents. This is debatable, and is what lawyers earn their money on.



Press Next.

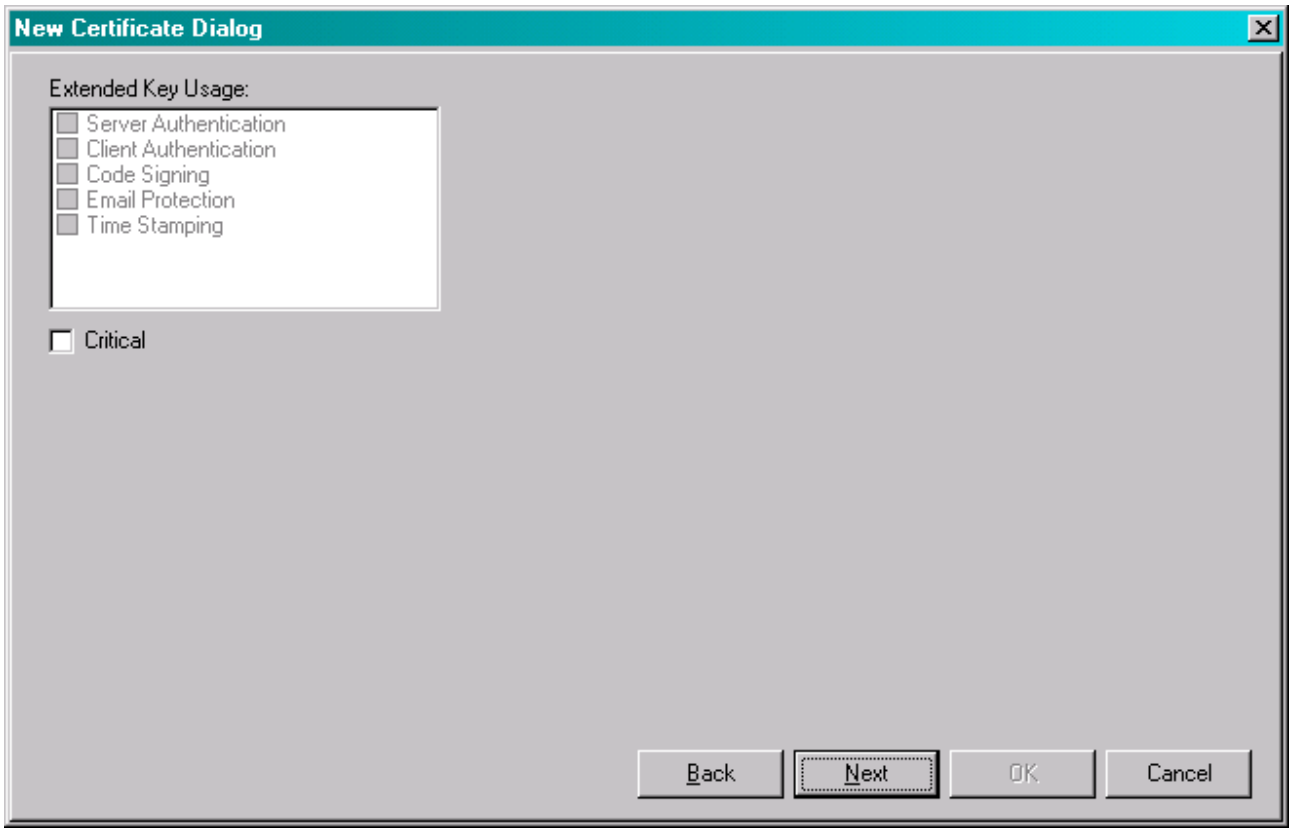
This page allow for you to specify how the keys in the certificate are to be used.
The keyCertSign is already checked, which is natural as your root certificate keys are going to be used for signing other certificates. This is after all the whole point with the root certificate.
You might want to check the cRLSign option if certificates signed by your root certificate should ever be possible to be revoked or invalidated by you. That way you can retract an already given client or server certificate. Leave the other options unchecked. Later on you will have to specify where revocation lists can be obtained.

More info about this can be found in the StreamSec documentation.



Press Next.

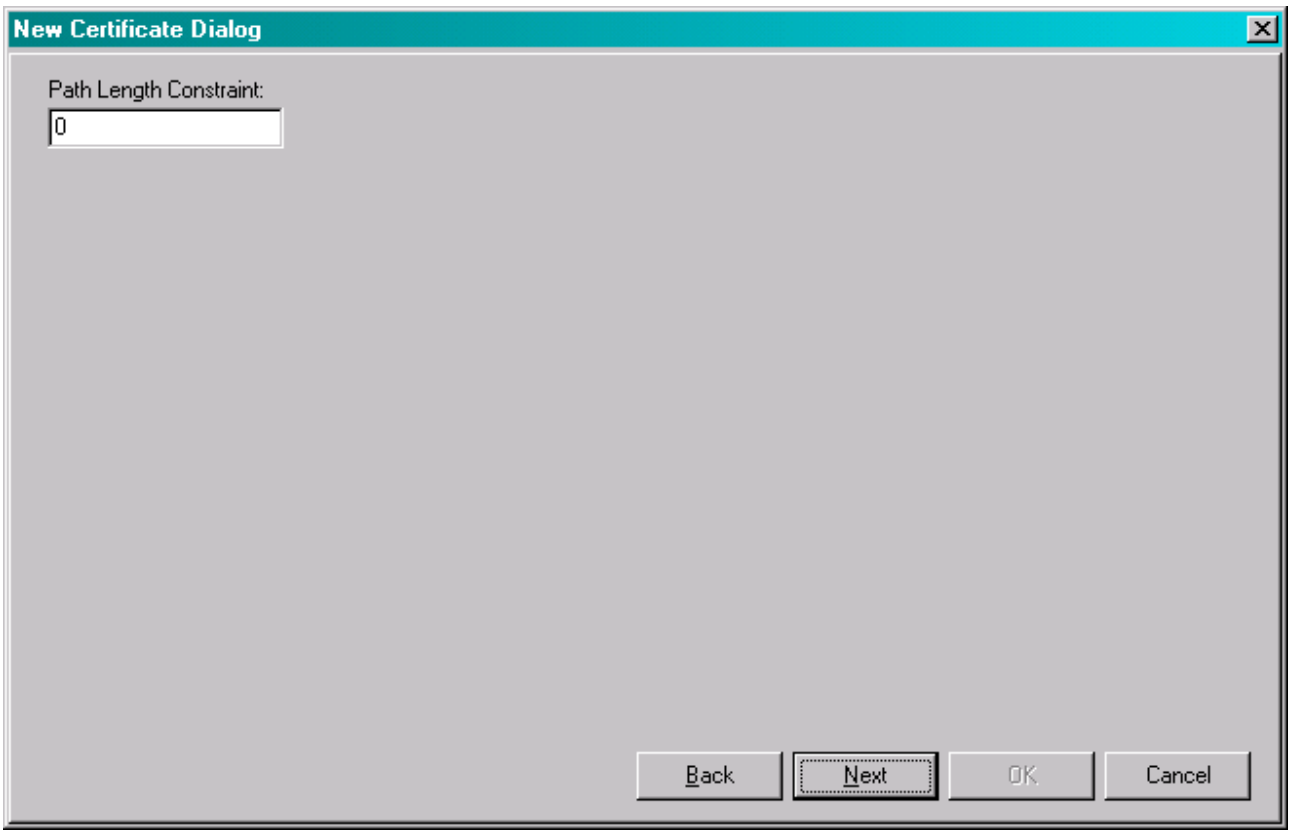
This page is used for selecting more specifically what type of operation the certificate is valid for. None of these options are available since none of them is consistent with the Key Usage (keyCertSign, cRLSign) you specified on the previous page.



Press Next.

This page allows for limiting the number of tiers of certificates that can be created underneath the root certificate. Think of certificates as a hierarchical tree with the top certificate being the root certificate. Specifying 0 as Path Length Constraint means that certificates signed by this root certificates cannot be used to sign other new certificates. Specifying 1 means that certificates signed by this root certificate can be used to sign one tier of certificates, but they can in turn not sign more certificates. Clearing the field removes any restriction.

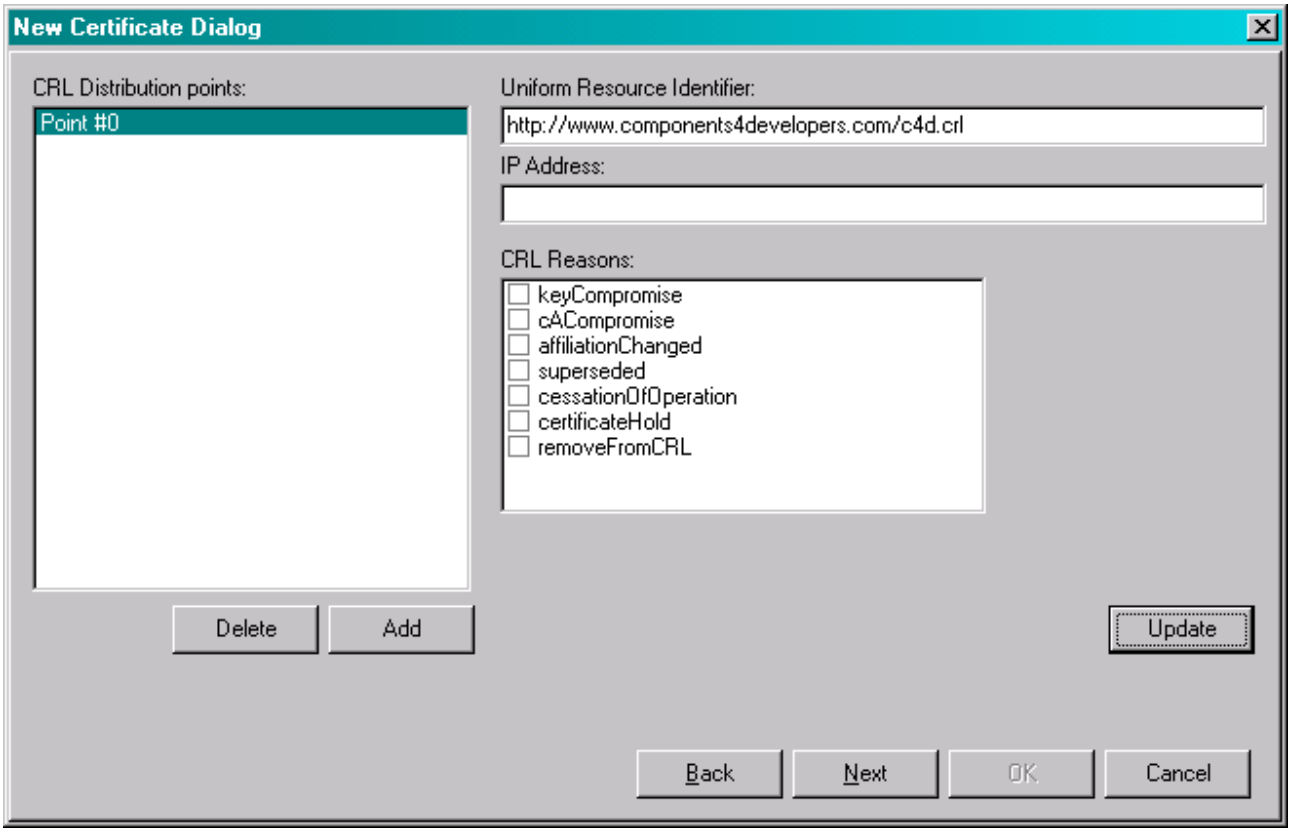
You will want to use the default value of 0 to be in fully charge of which certificates are signed with your root certificate.



Press Next.

Since we selected the option cRLsign on an earlier page, we opted to let certificates generated by this root certificate be revocable by the means of a certificate revocation list (CRL).

Thus the wizard displays a CRL definition page.



A CRL distribution point is typically a place on the web where *.crl files can be downloaded from. The *.crl file contain information about revoked certificates.
At least one distribution point must be specified if the cRLSign option was checked.
Each distribution point (which refers to the *.crl file) must have either a URI or an IP address defined. For our purposes we define an URI as shown in the dialog sample.

It is also possible for the CA to enter this information directly into the end-entity certificate. If the CRL distribution point information is not present in the end-entity certificate, StrSecII will assume the CRL is located at the distribution point specified in the issuer certificate.

Further it is possible to define CRL reasons. If any reasons are checked, it means that that specific *.crl file only contains certificate revocations due to those checked reasons.
This way it’s possible to distribute certification revocation lists over multiple sites, each taking care of specific reasons.
If no reason is given, the distribution point must contain all reasons.

Note: StrSecII will not download the CRL. When you write a certificate using application, you must add code that downloads the CRL and adds it to the SimpleTLSInternalServer component that is using the certificate in question.

The following reasons are supported by StreamSecII:

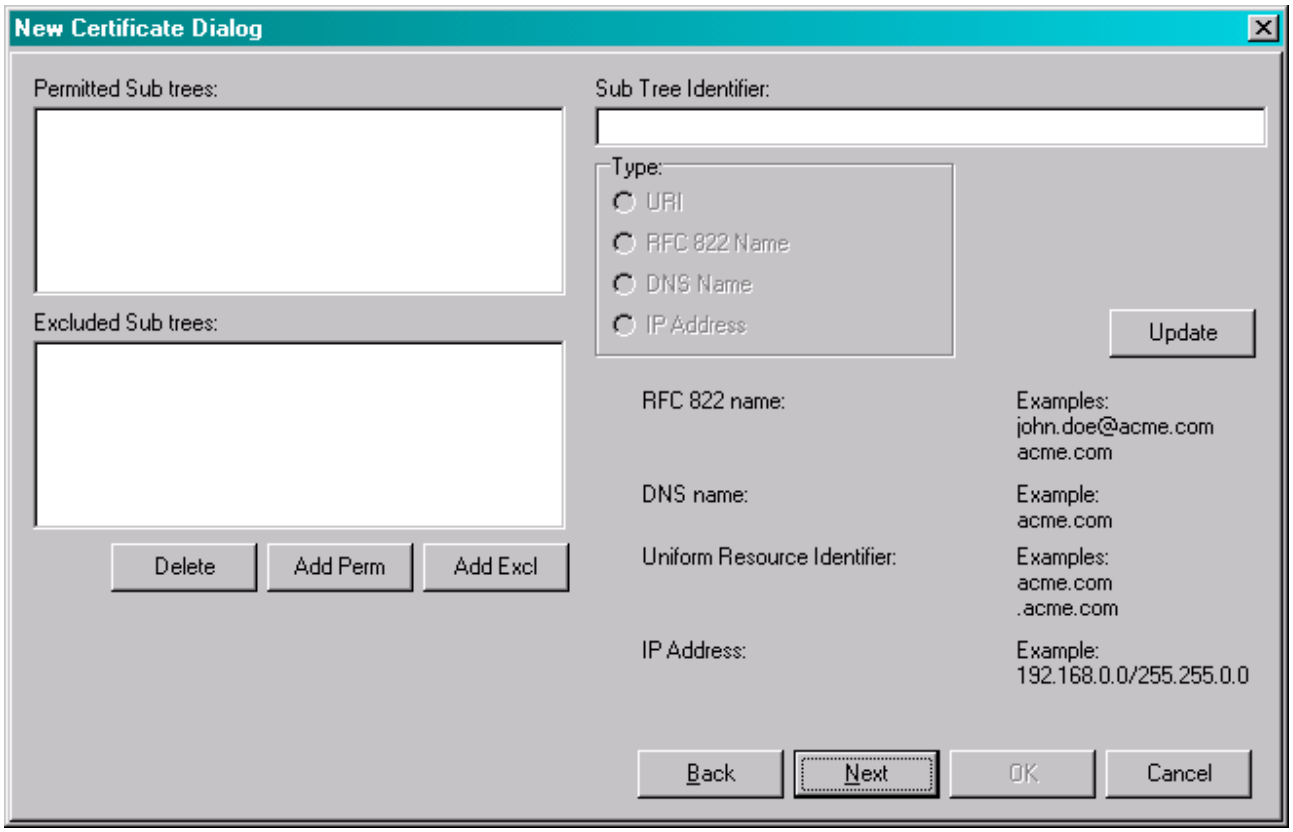
keyCompromise	Someone who shouldn't have might have apprehended the private key corresponding to the certificate.
caCompromise	The CA (root certificate) has been revoked with a keyCompromise. All issued certificates should in such case be revoked with this code.
affiliationChanged	This code is suitable if an employee has an emailProtection or clientAuthentication certificate, and the employee quits, gets fired, transferred to another unit etc.
superseded	A newer certificate for the subject exists.
cessationOfOperation	This code is used for web server certificates when the server/domain is shut down.
removeFromCRL	Should be avoided. Means that the certificate has been previously revoked and that was a mistake. I guess this is mainly used to undo certificateHold revokations.

Since most relatively small setups will only have/need one entry in the CRL distribution list, don't select any of the reasons to allow for them all to be fetched from that single distribution point.

Thus add a distribution point by clicking Add.
Make sure to fill out the URI.
Leave the reason options unchecked.
Click Update.

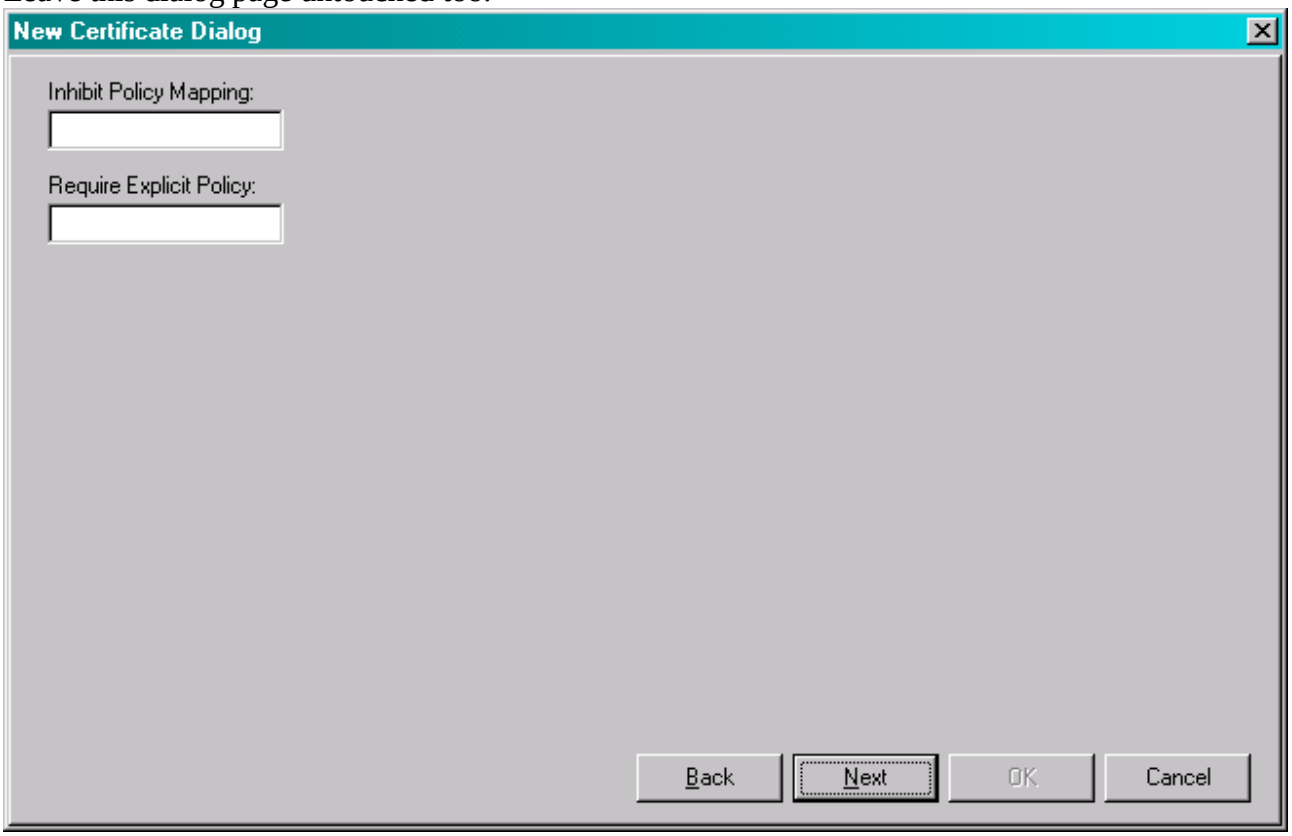
Press Next.

Leave next dialog untouched. These options are mostly useful to set in intermediary CA certificates, when you want to restrict who might get a valid certificate issued by such intermediary CAs.

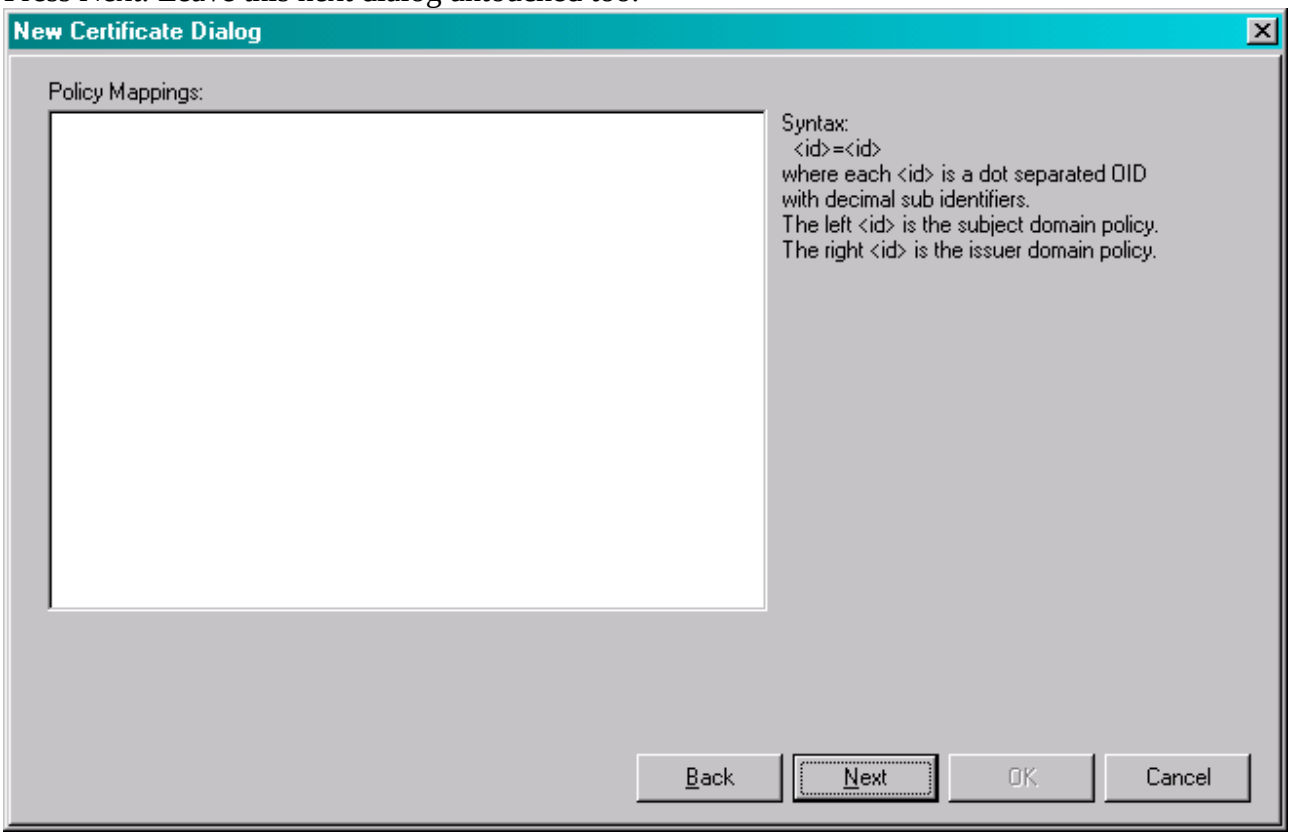


Press Next.

Leave this dialog page untouched too.

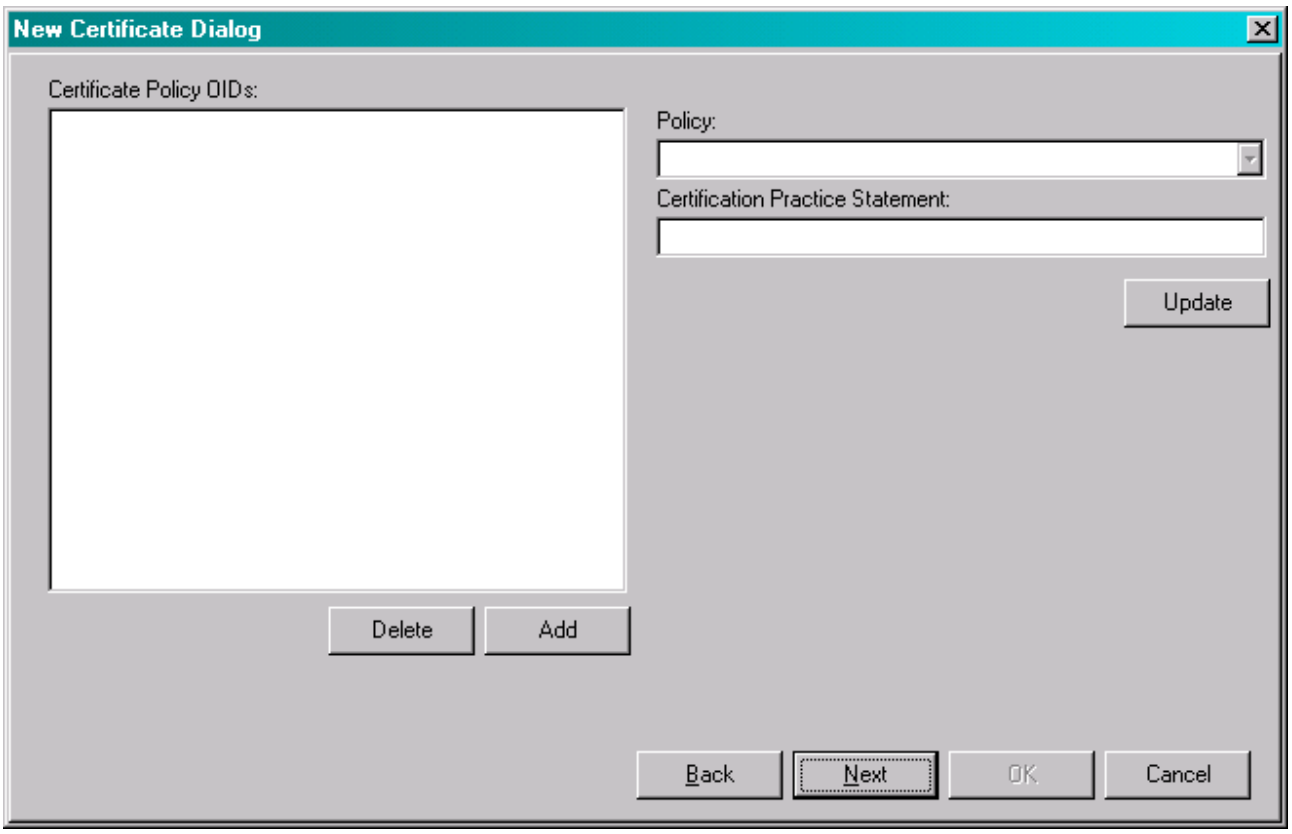


Press Next. Leave this next dialog untouched too.



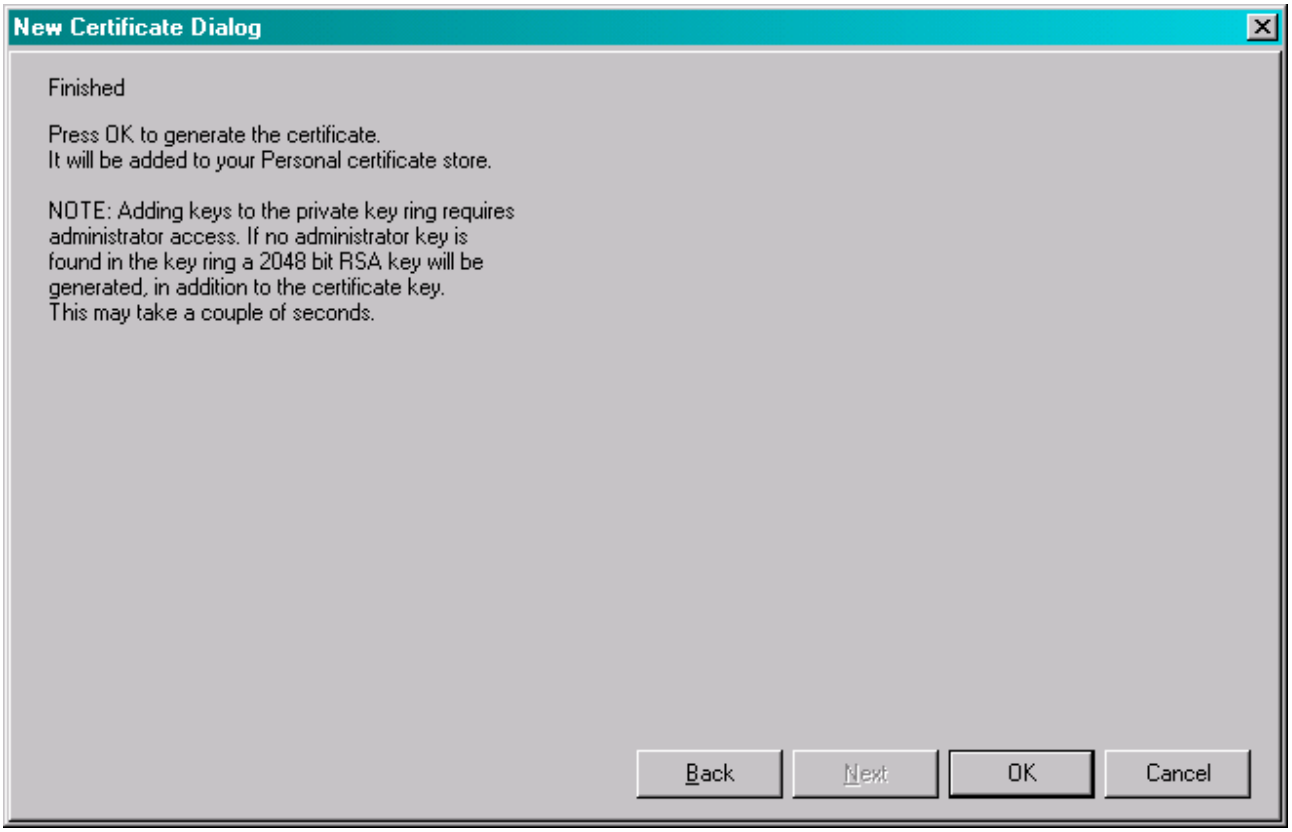
Press Next.

This dialog page can be left untouched. Optionally, you might write a document describing e.g. your security policy and the intended purpose of the certificates issued with this root certificate. Click Add. Enter the URI of the document in the Certificate Practice Statement field. Click Update.

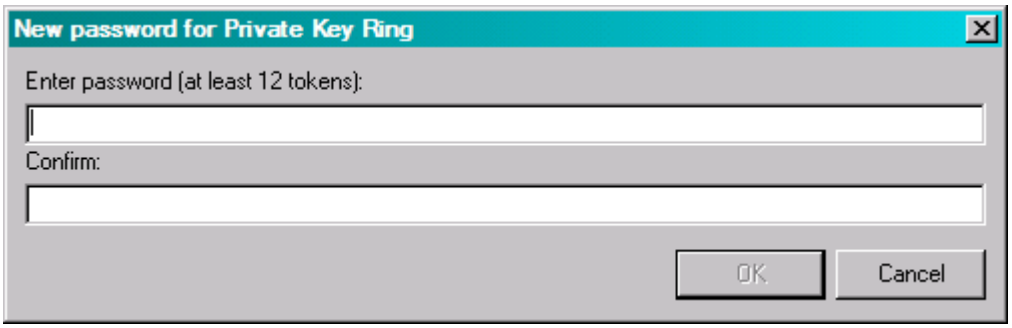


Press Next.

And then the final dialog page.



Click OK. This will start to generate the public and private keys for the root certificate.
Because the root certificates private key ring should be protected, a dialog will now pop up asking for entering a password.

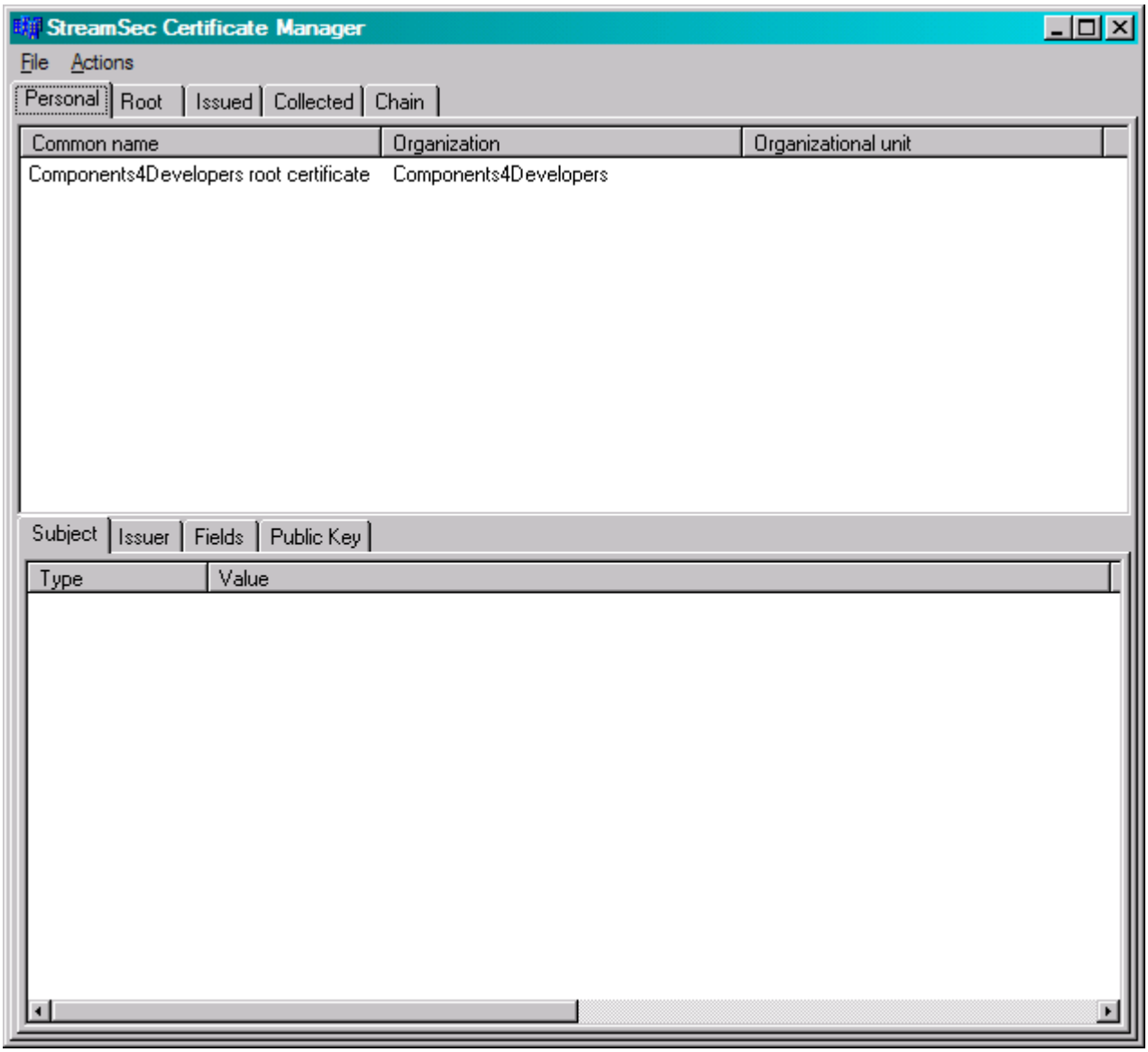


Enter a password you can remember of min. 12 characters length. If you loose this password, you will loose read access to your private root key. This would make your root certificate unusable for creating new certificates.

Next it will ask for the admin password for the private key ring. The admin access to the key ring is needed when the keys in the key ring are going to be updated or altered or new private keys going to be added.

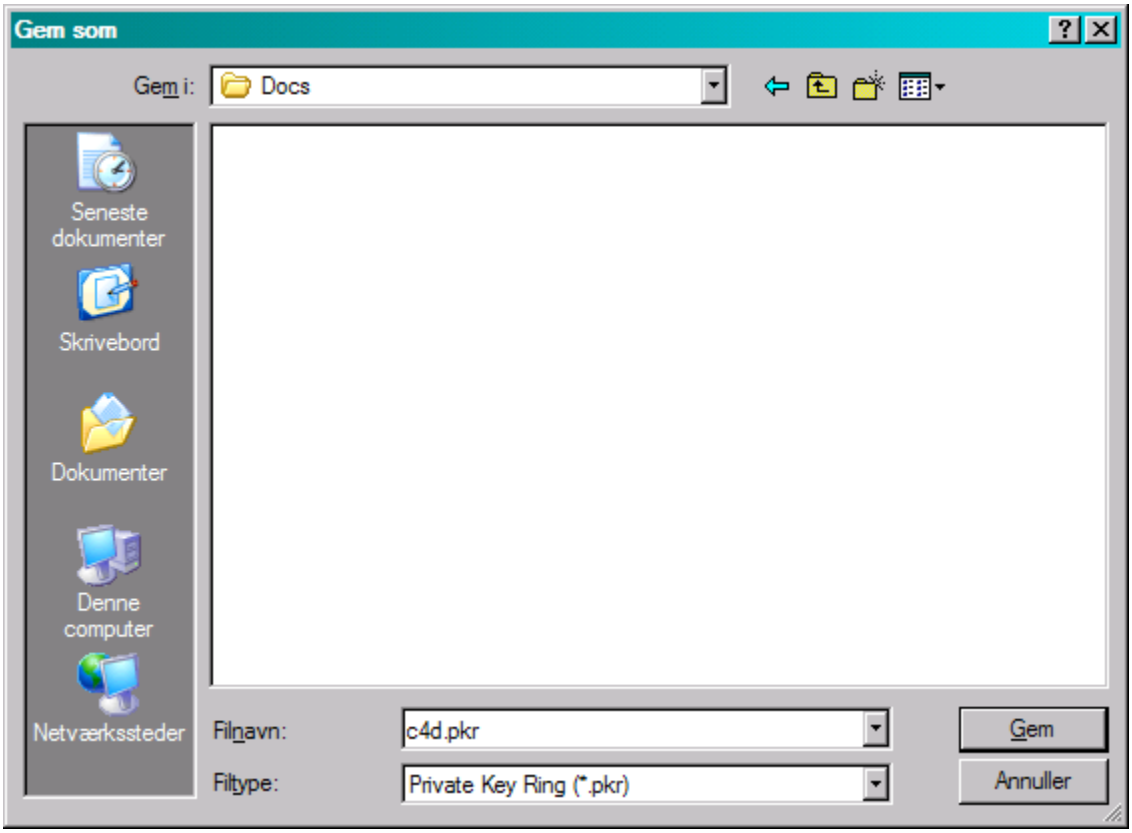
The password can be of any length as long its more than one character. You also need to store this password in a very secure place since it's the one permitting changes to your private key ring.

After providing the passwords, the private and public keys are calculated and the root certificate generated.



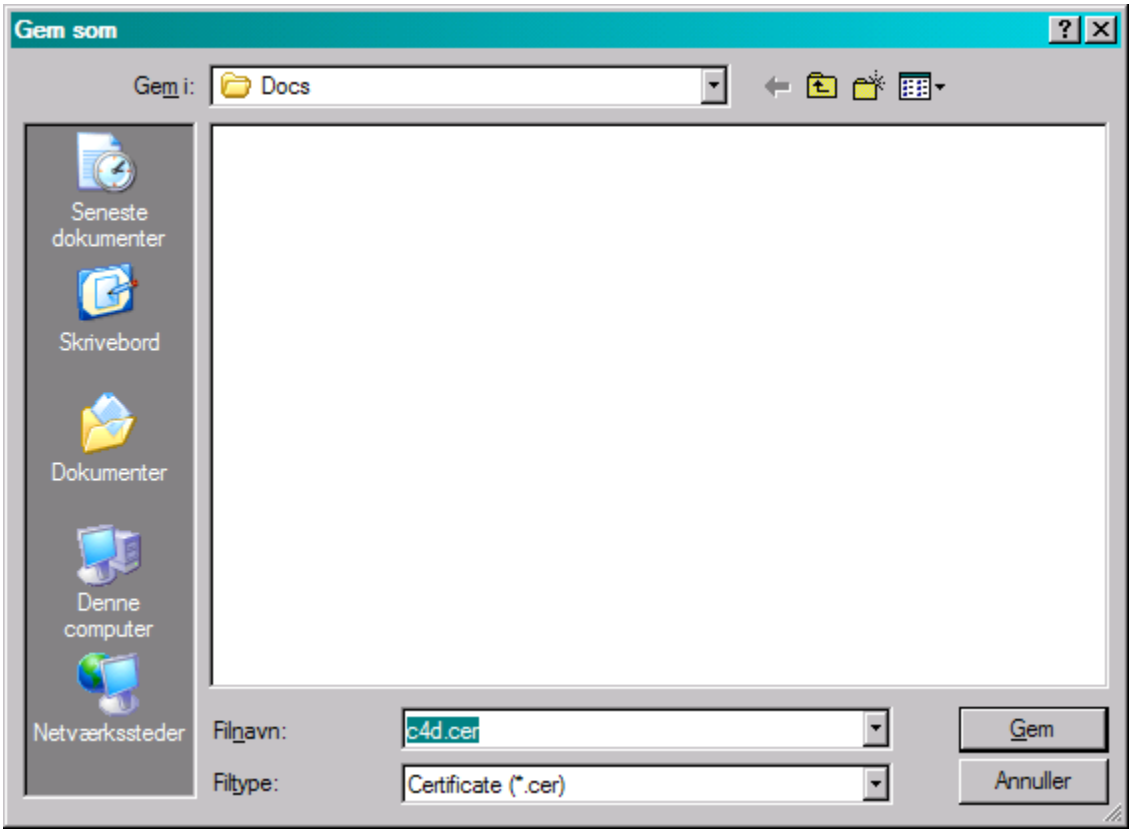
Save the private key ring containing the private key for the root certificate in a file and store it in a secure place. Other people must never be able to obtain your private key. If they do, the security of your root certificate has been severely breached.

Click File/Save As to save your private key ring.
Remember to add the extension .pkr to the filename you type.



Specify read access password for the key ring file. Presently this password MUST be identical to the read access password you specified when the public/private key pair was created.

Next save the public part of your root certificate.
Select File/Save As and file type Certificate (*.cer).



Remember to add the extension .cer to the filename you choose.

Now also the public part of the root certificate has been stored. These two files together forms a complete root certificate and can be reloaded later on if the you should loose the settings in the CertMgr applications for example because of a HD crash etc.

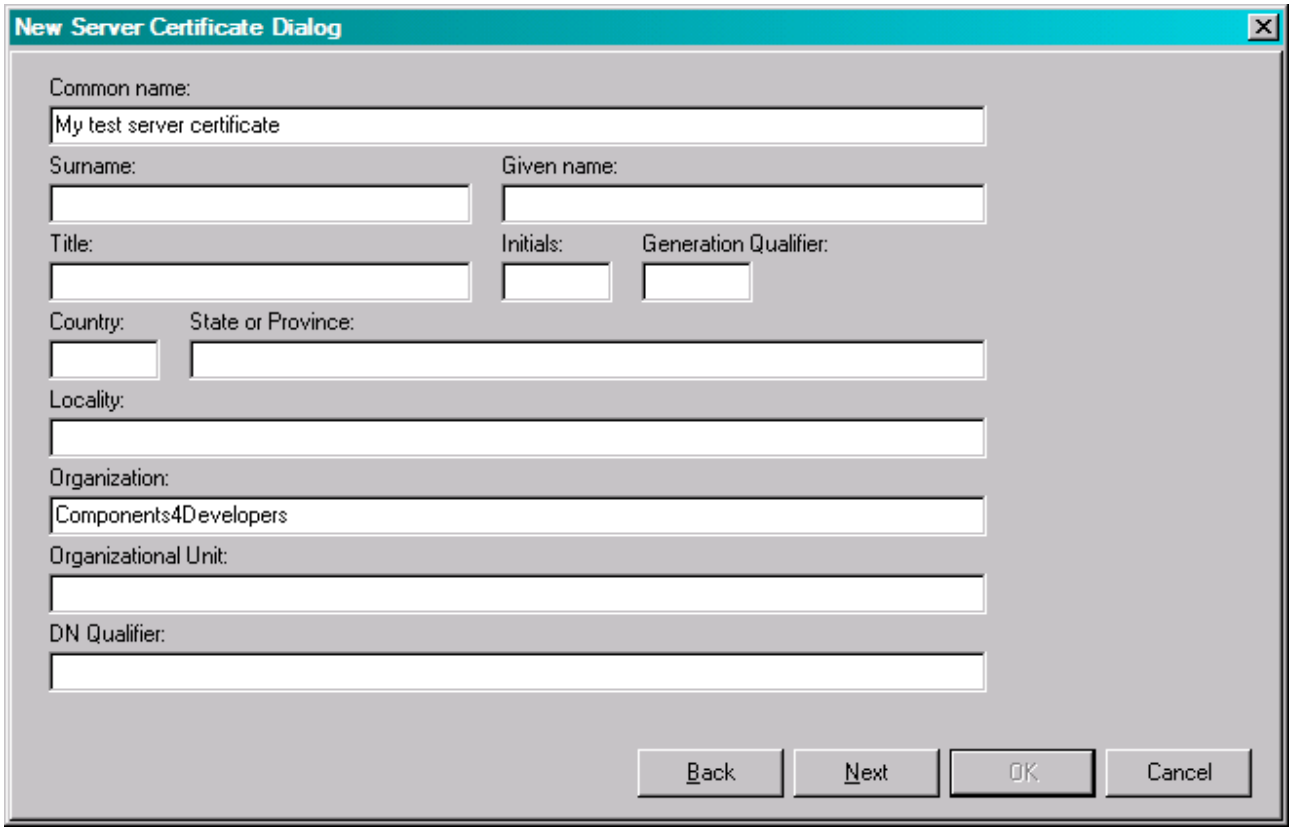
Next step is to create a server certificate that is signed by the root certificate you have just created.

It's quite close to the same method as creating a root certificate but with a few twists.

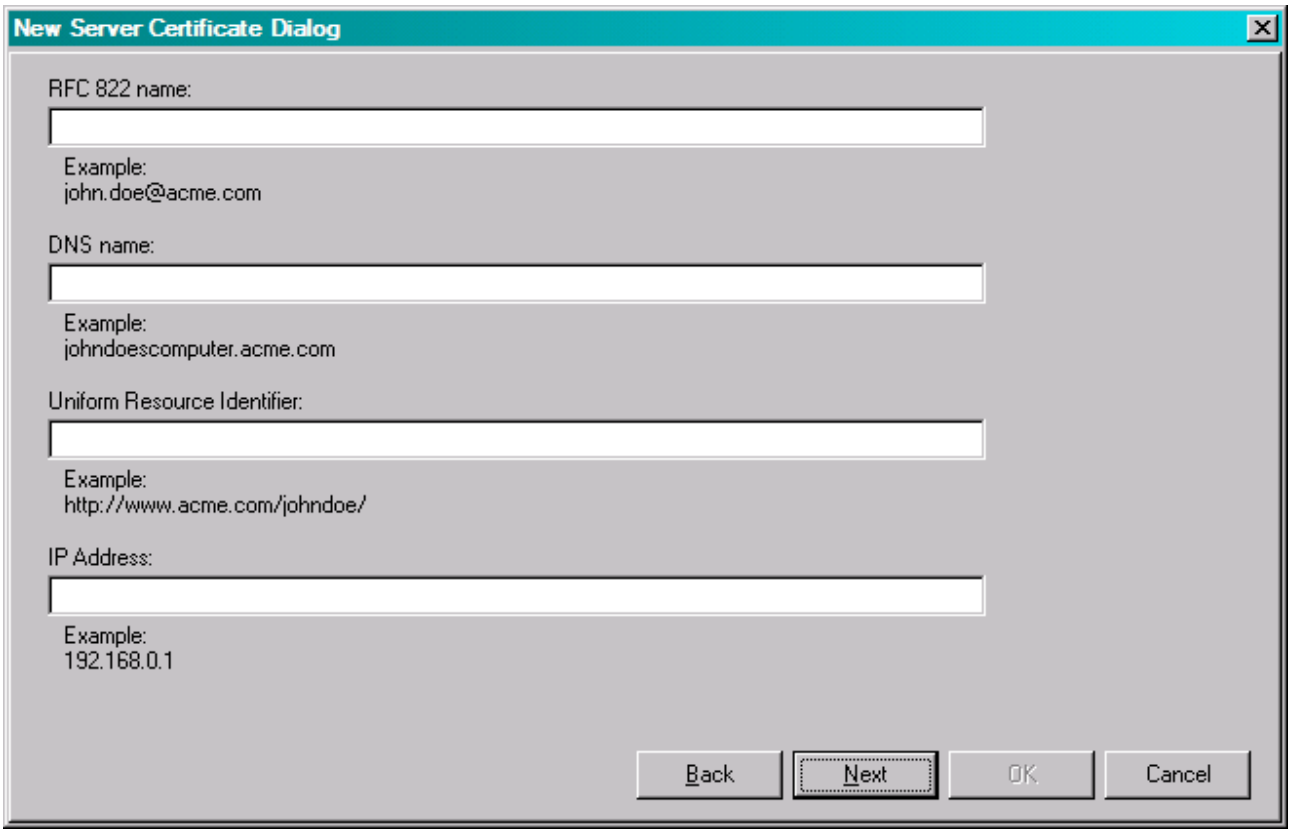
Import your root certificate by right clicking on the TSimpleTLSInternalServer component and selecting 'Open File'. This displays an Open file dialog. Open the *.cer file you saved a couple of steps back.

Next step is creation of a server certificate signed by your root certificate.
Right click the TSimpleTLSInternalServer component and select 'Create Server Certificate Request'.

It shows a dialog similar to when we created a root certificate. This time you should fill out information matching the user of your server certificate. Something must be filled into some of the fields, but you are free to choose what.

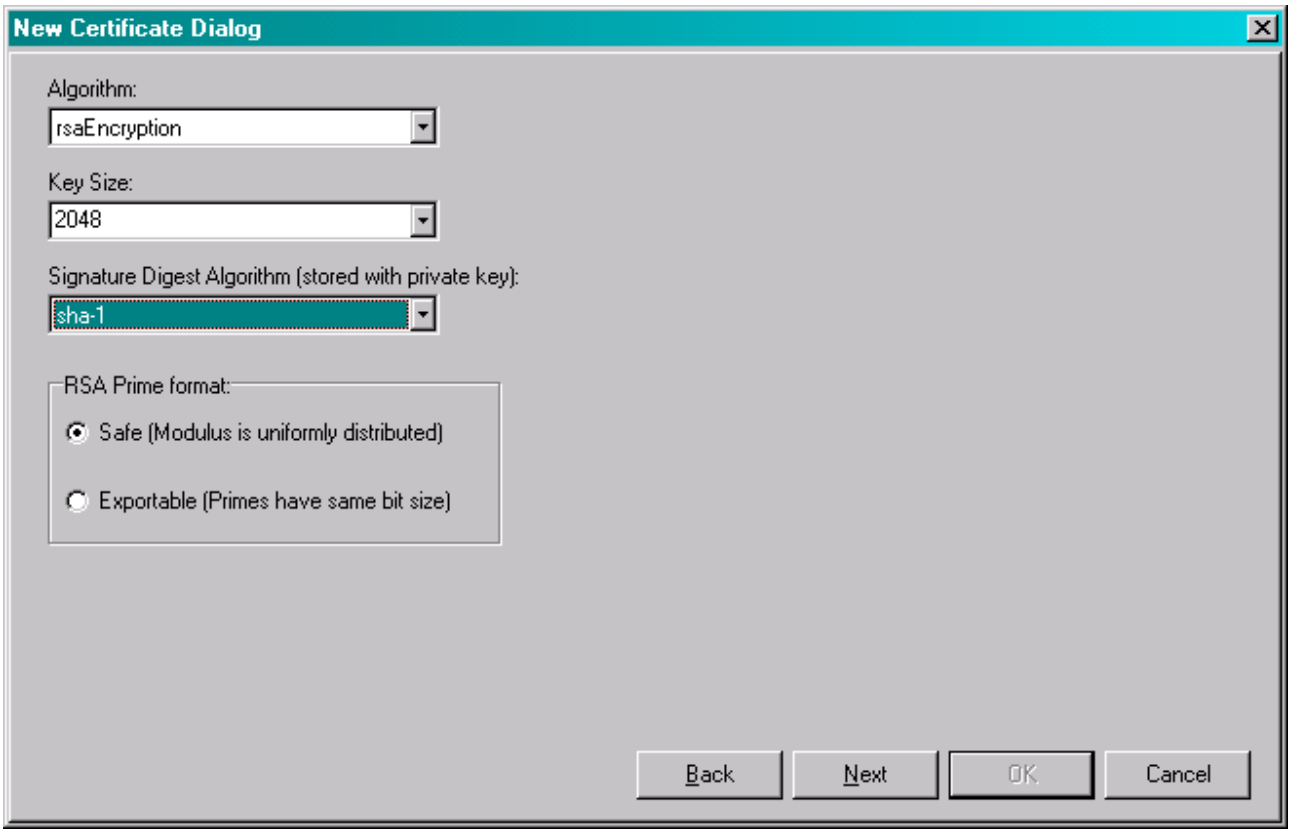


Press Next.

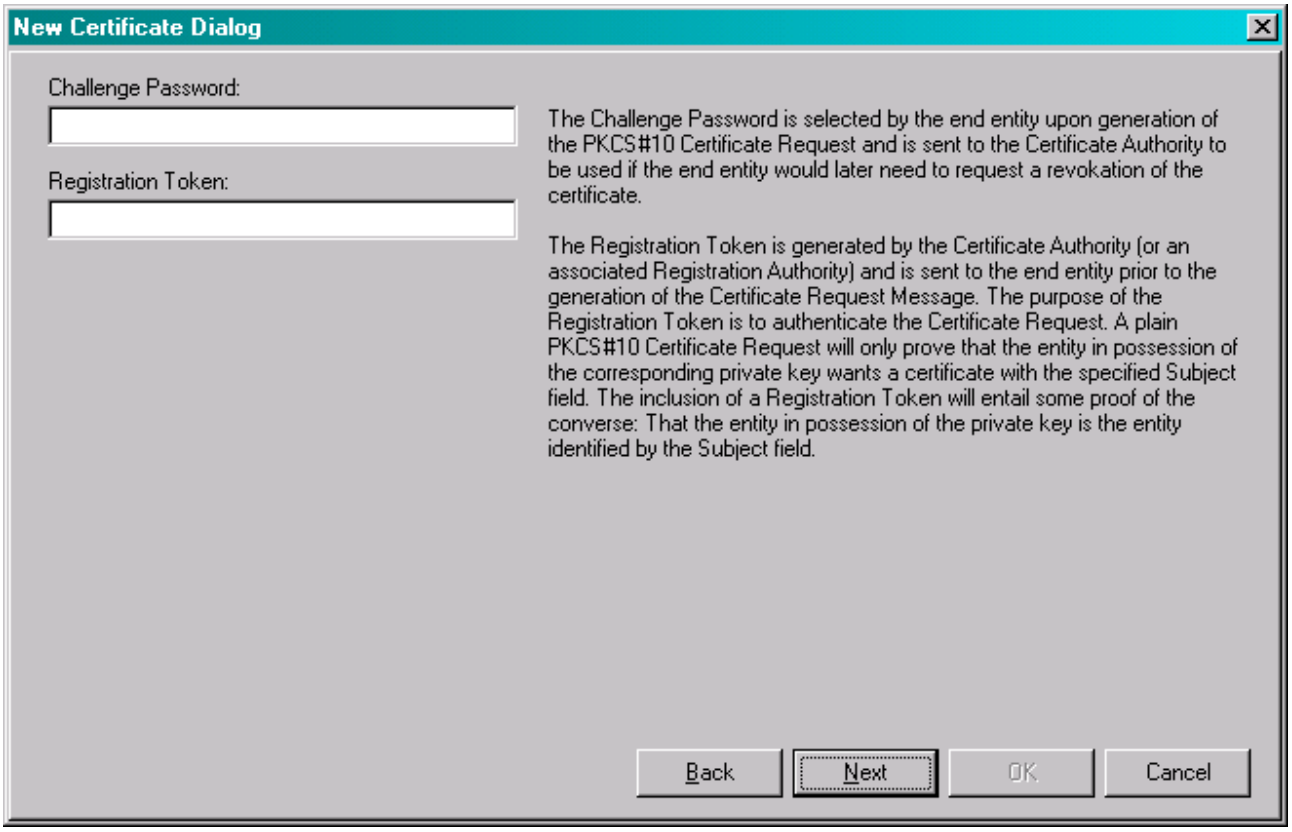


If you want to bind the server certificate to a specific server, then fill out the DNS name or IP address.

Press Next.

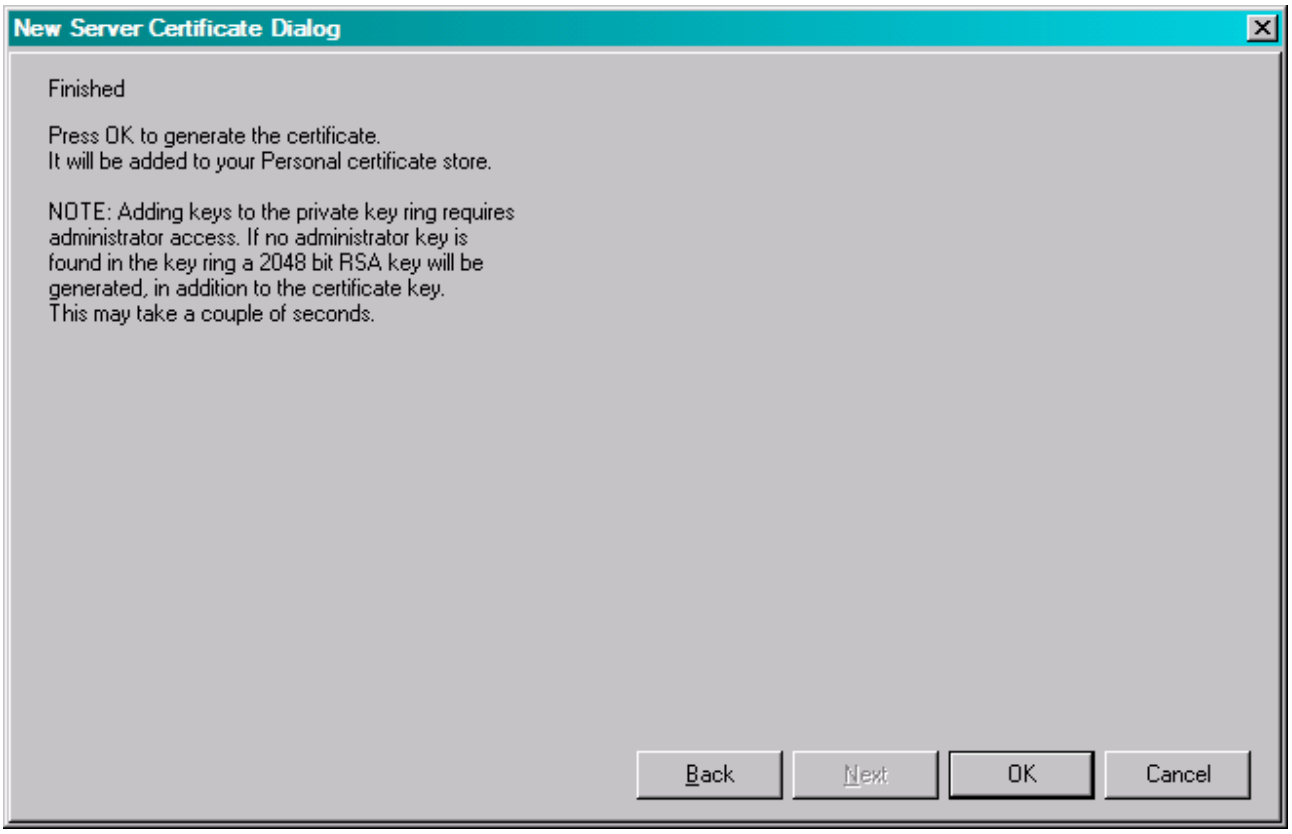


Again select the encryption method you want to use, but now for the server certificate.
Press next.
The next page is the CRL distribution point page. Leave it blank. Press next.

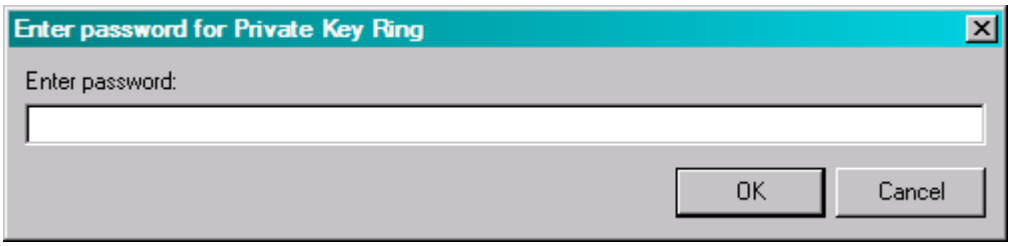


You don't have to specify these fields since you are in control of both the root certificate and this certificate. These fields are used if you don't have any other means to authenticate yourself to the Certificate Authority, but in this case you are your own Certificate Authority.

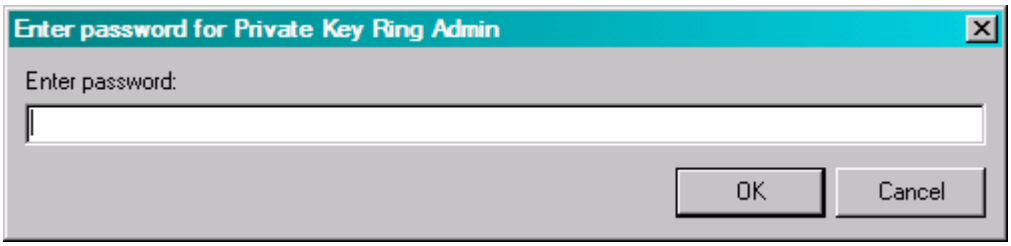
Press Next.



Press ok. Since it's the first private key you are generating for the private key ring administered by this instance of the TSimpleTLSInternalServer component, you must choose two passwords, one for read access and another for administrative access to the key ring. The passwords protect the keys in the key ring (in the component) so nobody but the people knowing the passwords can get information about them or change them. It is of vital importance that you remember these passwords and store them in a safe place. If you loose them, you will not be able to access your certificates any longer.



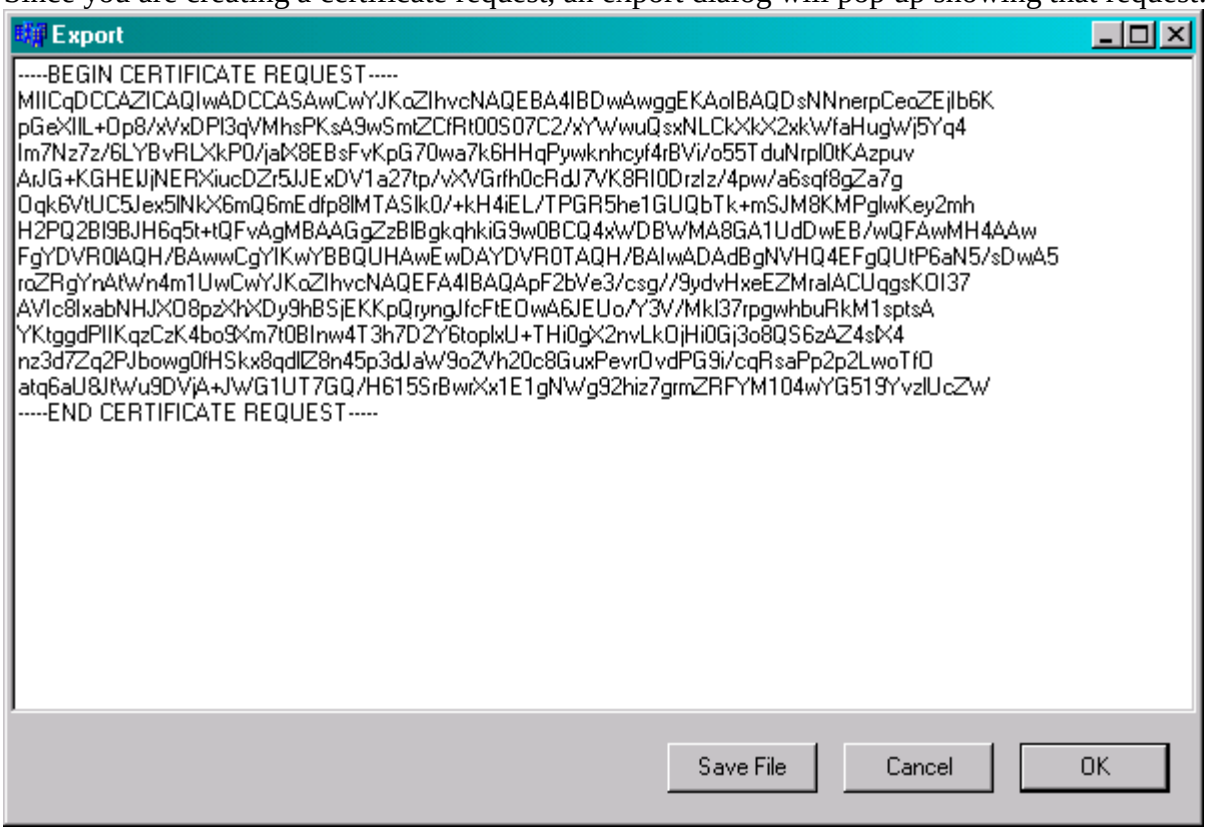
Enter a password (for example ABC). Then press ok and enter the administrative password:



Choose another password for administration than for read access, for example ‘RoastingBananas’. It’s important not to put spaces in the passwords. If you do make a mistake here, and press OK, you will have to start all over again.

After a short time where the key calculations are happening, the root certificate and its public and private keys have been generated and stored in that components key ring.

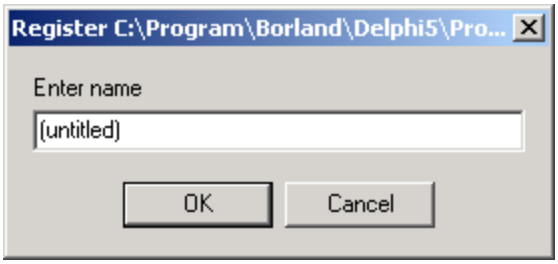
Since you are creating a certificate request, an export dialog will pop up showing that request:



This certificate request can be copied from the memo, or saved in a binary format to a *.p10 file. Click OK when you done that.

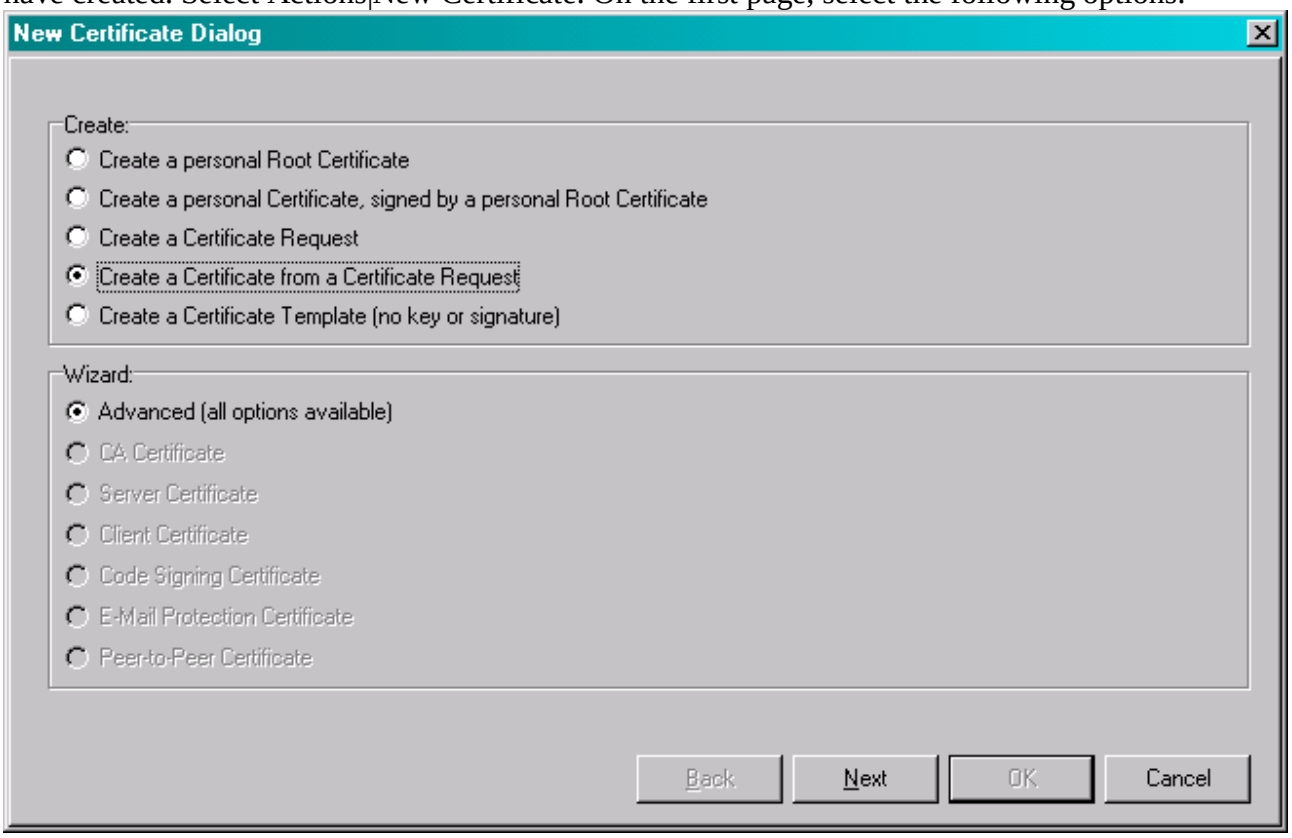
Next, a Save dialog will pop up prompting you to save the private key ring to a file. It is good practice to do so for back up purposes. The private key ring you save this way has to be named User.pkr, so create a new directory if necessary. (This is a feature added in StrSecII 1.8.146)

Next, an InputQuery dialog pops up:

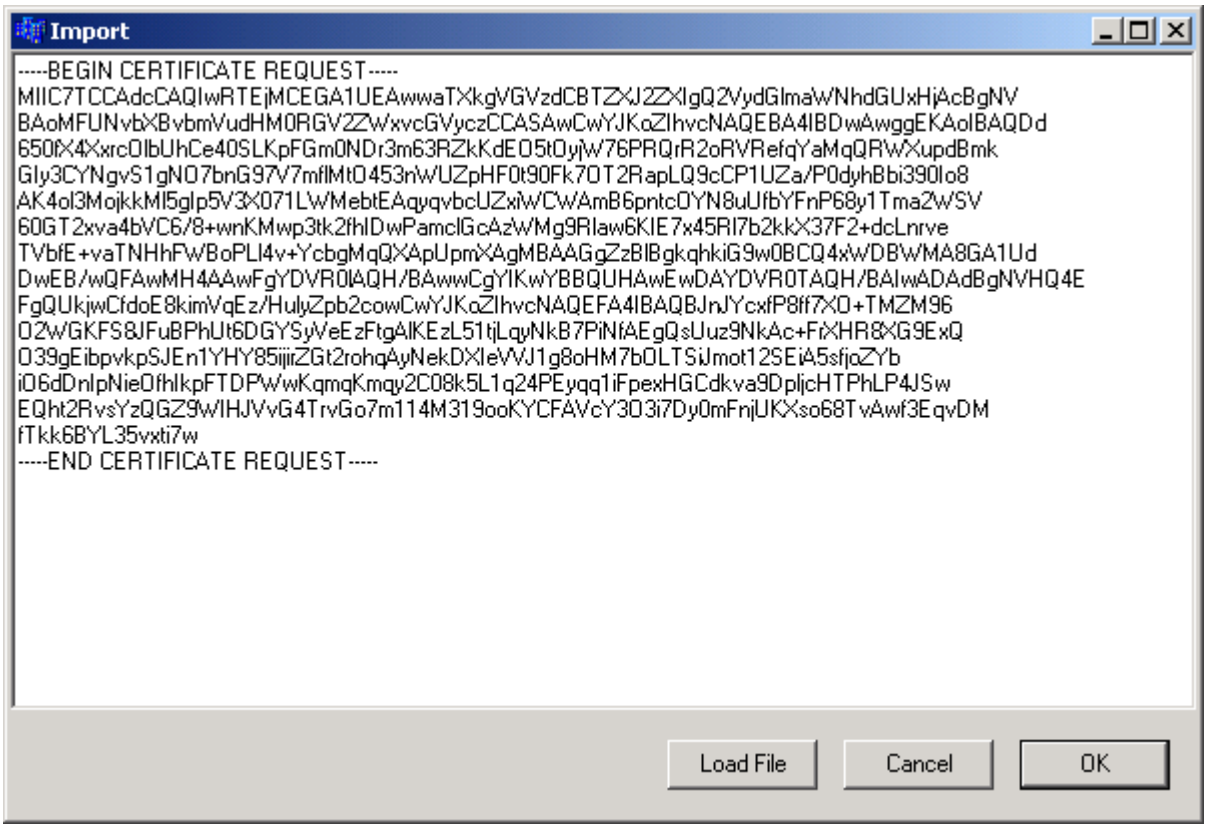


This query will prompt you to register the folder where you stored User.pkr. Press Cancel at this point. (This is a feature added in StrSecII 1.8.146)

You should now turn back to CertMgr and issue a certificate based on the certificate request you have created. Select Actions|New Certificate. On the first page, select the following options:



Click Next.
An Import dialog will pop up. Paste the certificate request in the memo field:

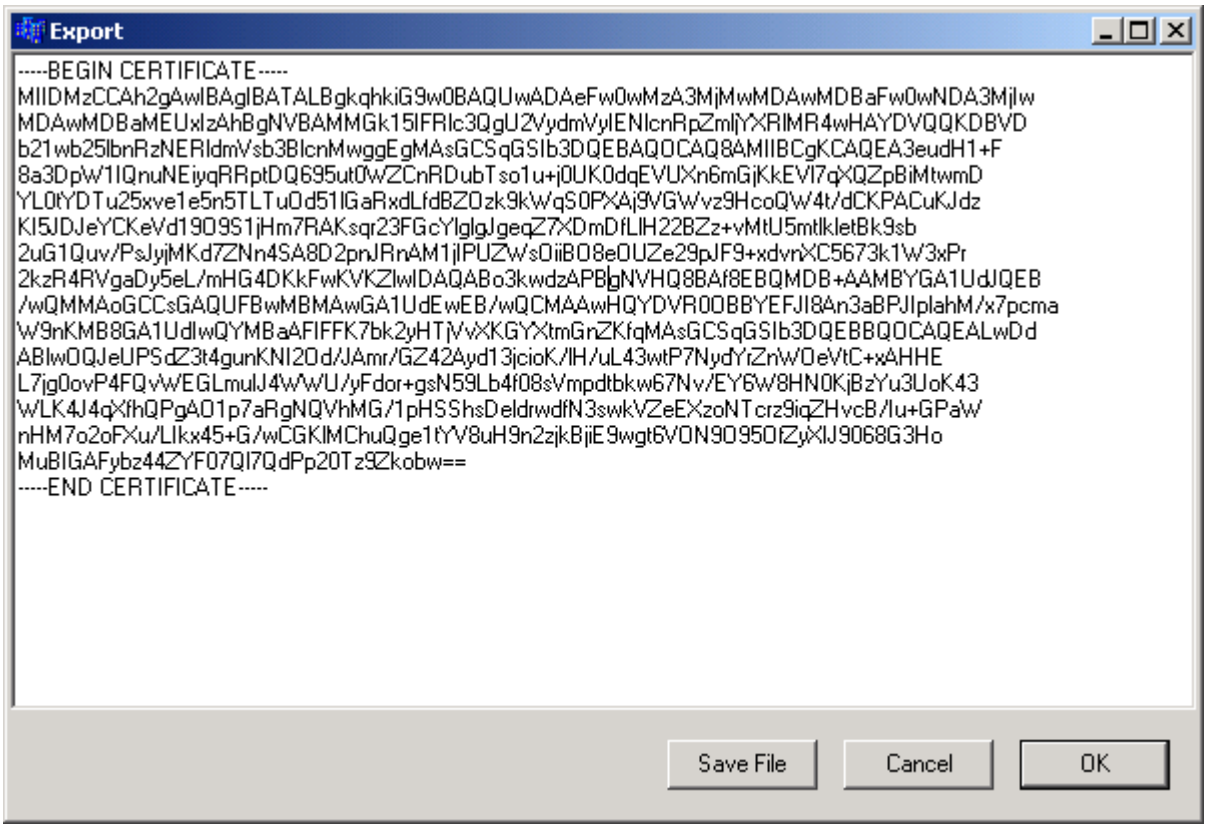


Click OK.
You may now click your way through the rest of the dialog. Most of the pages will just display the information you entered in the design time wizard for the TSimpleTLSInternalServer component.

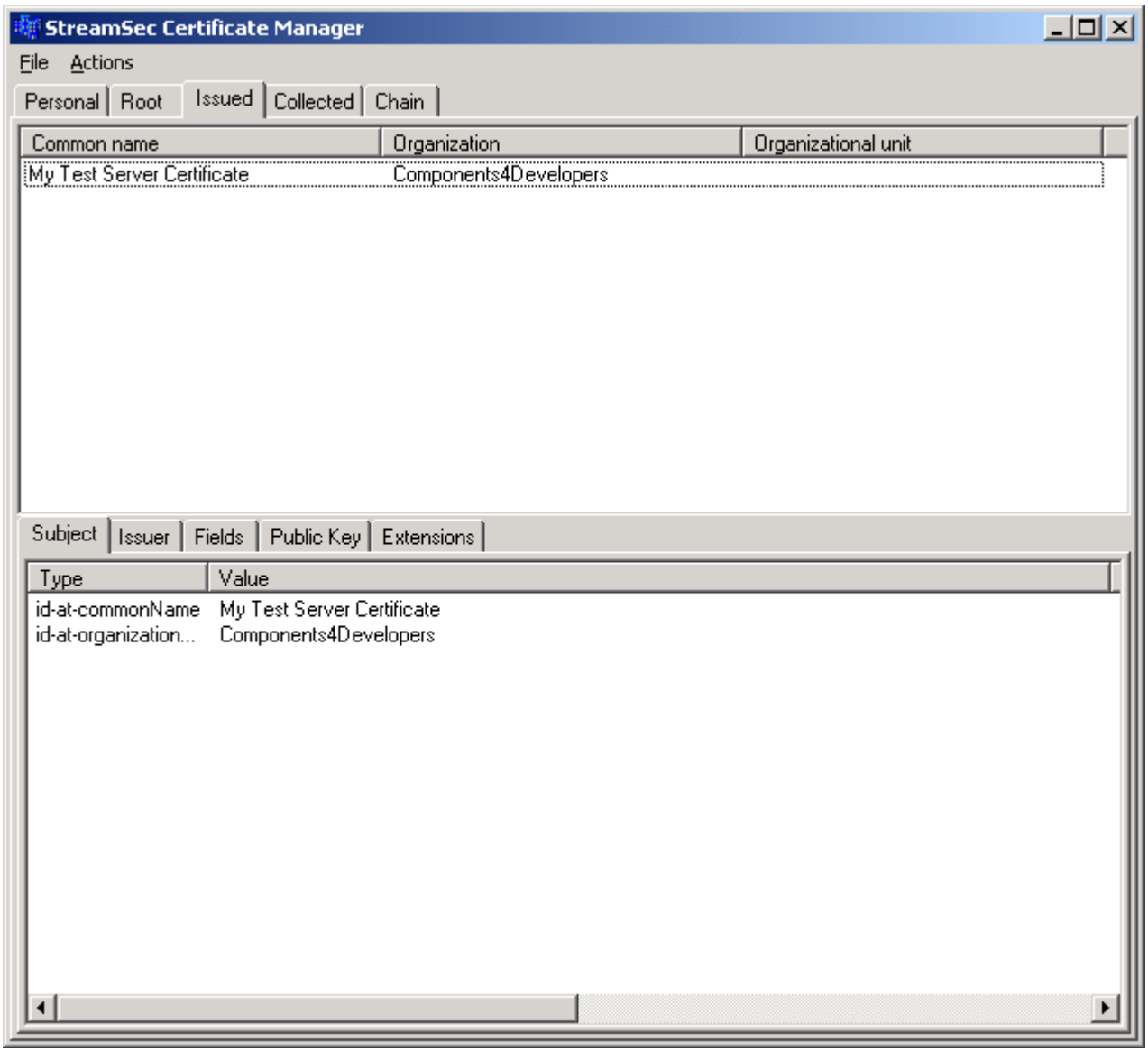
Two pages are however new: The Issuer page, where you currently only have one option since you only have one CA certificate, and the Validity page where you set the validity period of the certificate. The default validity period is one year, and it must be completely contained in the validity period of the issuer certificate(s).

You may also change any information (except the public key) at this point. If someone else had generated the certificate request you would have to verify the authenticity of the information, and possibly add your own CRL distribution point and certificate policy information. This would also be the point when you would verify the Registration Token.

When the certificate has been signed, another Export dialog will pop up, this time showing the signed certificate.

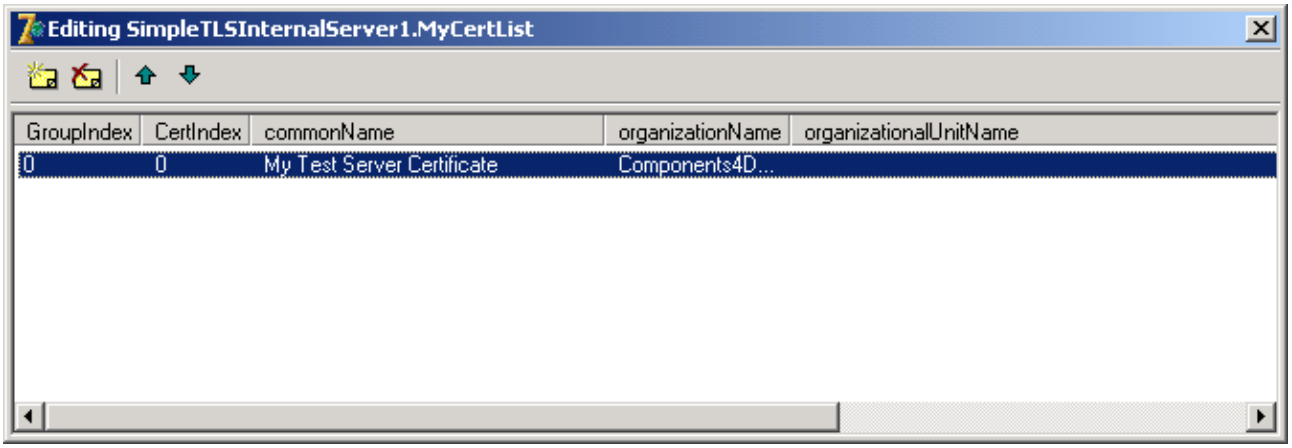


Click Save File to save the certificate to a *.cer file. Click OK. The new certificate will be stored in the Issued store of CertMgr, as seen below:

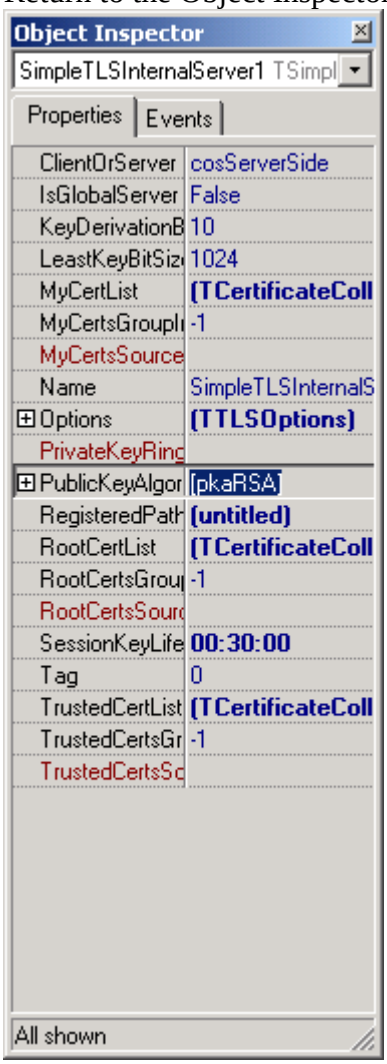


Return to Delphi and your TSimpleTLSInternalServer component. Right click and select Open File. Select the Personal Certificate File filter and the *.cer file you just saved from the Export dialog of CertMgr.

To check that the certificate was loaded properly, double click on the MyCertList property of the TSimpleTLSInternalServer component, and select the Add button in the collection editor.



The certificate is there!
Return to the Object Inspector:

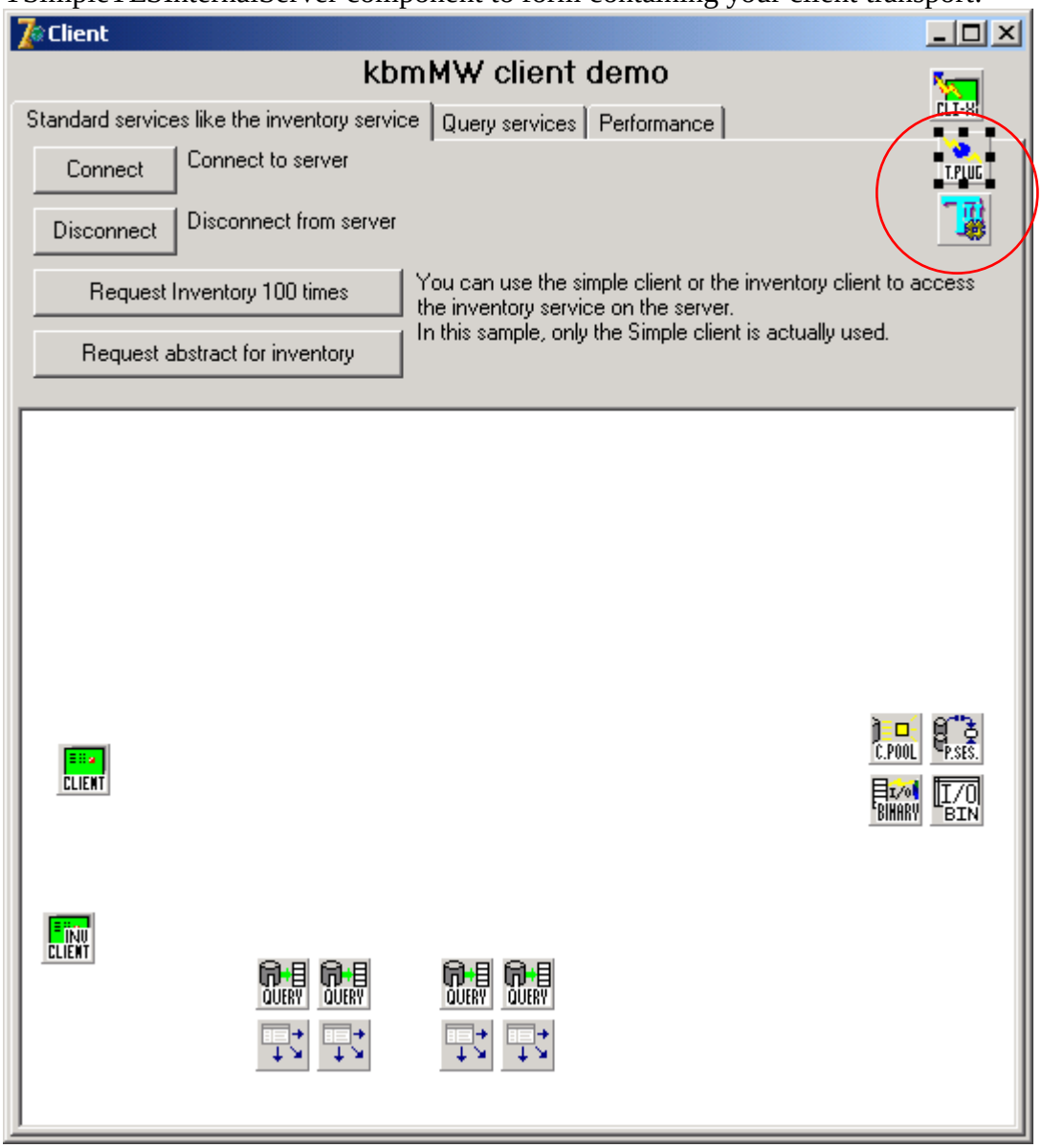


Another indication that every thing is working is that the PublicKeyAlgorithms property changed to [pkaRSA], indicating that you have a server certificate with a RSA key that can be used for SSL/TLS.

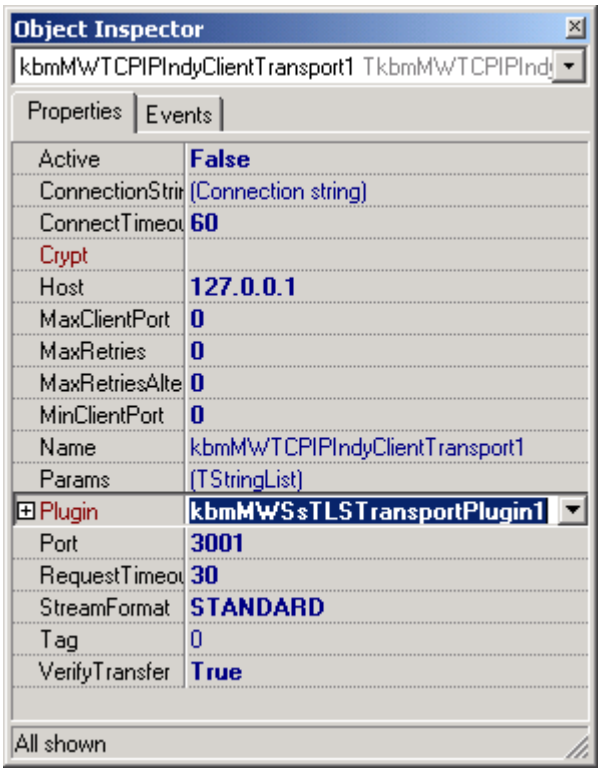
I will suggest you to save your project now, close it and then reopen it. It will ask for your read password to the key ring when opening the project.

How to secure an existing application client?

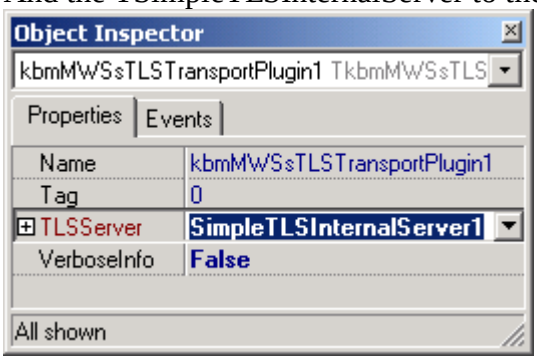
Just as with the server, simply add a TkbmMWSsTLSTransportPlugin and a TSimpleTLSInternalServer component to form containing your client transport:



These components have to be linked in a fashion similar to the way they were linked on the server form. Link the TkbmMWSsTLSTransportPlugin to the TkbmMWTCPIPClientTransport's Plugin property.

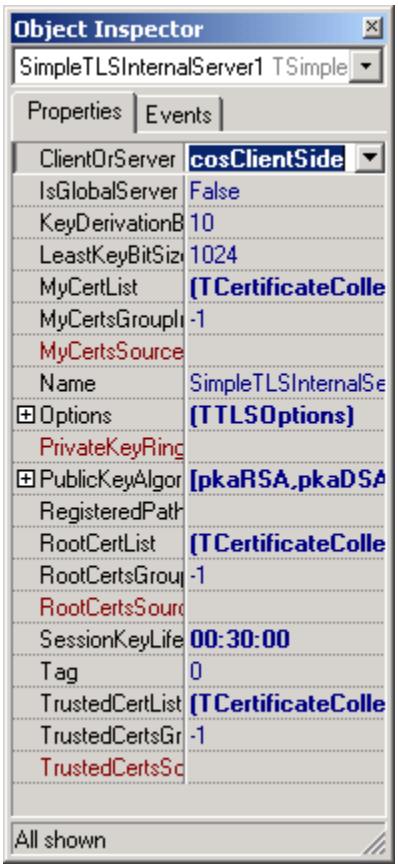


And the TSimpleTLSInternalServer to the TkbmMWSsTLSTransportPlugin's TLSServer property.



Next step is to import the root certificate that was used to sign the server certificate. Right click on the TSimpleTLSInternalServer component, select Open File and the Root Certificate File filter. To prevent format problems it is recommended that you open a *.scl file containing the root certificate. If you have closed CertMgr, the file User.scl will already have been saved to the path of CertMgr.exe. Otherwise you may create it from the running CertMgr application by selecting File|Save As and the Signed Certificate List filter. Enter the filename 'User.scl' and click Save. A dialog will pop up where you should select the My Certificates option. Click OK.

You must also change the ClientOrServer property of the TSimpleTLSInternalServer component:



Note that by changing the ClientOrServer property the PublicKeyAlgorithms property changed to include the full set of supported public key algorithms. This is because with SSL/TLS the client doesn't need a private key and certificate, unless the server request one and aborts if the client fails to produce it. Most HTTPS servers do not require client certificates (because most users don't own one), but with kbmMW it is both feasible and recommended for the server to request and require client certificates. This means that you must add a method for the client user to obtain a certificate that can be used for client authentication with your server.

At this point you should do a basic security analysis of your application, in order to determine which is the most practical and most secure way to get the client certificates to your users. Some of the questions you should try to answer are:

1. What kind of identities are the certificates meant to authenticate? Possible answers are:
    a. Physical or "real life" identities. This is typically the case with corporate servers and when the client users are employees. You want to ensure that the person operating the client software is exactly that flesh and blood person the certificate says it is. The solution in this case might be to exchange certificate requests and certificates on 3.5" disks handed over in person.
    b. "Commercial" identities, e.g. that the person or entity in possession of the private key is identical to, or at least trusted by, the person or entity that purchased your software on your e-commerce site. The solution in this case might be to display a Registration Token on the HTTPS site once the transaction is completed. The client enters the registration token into the certification request, and the server verifies that

the registration token and the subject name match the delivery details of the original order before signing the certificate.

   c. "Internet" identities, such as players who connect to a game server. If your only interest is to ensure that the client who registered as "Killer" last month is identical to the client who tries to log in as "Killer" today, then certificate issuance is unproblematic: The client simply registers by generating a certification request, and all the server has to do before automatically signing the certificate is to check that the name hasn't already been registered.

2. What will happen if the client private keys get compromised? Ideally, you want three things:
   a. Minimize the damage such events have on your own part of the system. This is mostly a question of common sense. A user doesn't get more reliable and trustworthy just because you have given him or her a certificate. It is (of course) not safe to give anyone administrator privileges just because the connection is protected by SSL with client authentication.
   b. Give the clients strong incentives to prevent key compromises from happening. You should have both game theory and Bayesian decision-making in mind when you design the application. Inform both your self and your users about your mutual interests. Never build a system where your own security relies on the assumption that your users will perform some critical task they have no informed self-interest in doing. They won't. Keep this in mind in particular if the answer to question 1 is b. One of the most common forms of piracy is to purchase a single license of the software and share it between several users. In other words, it is bad security to assume that any commercial identity is equivalent to a single physical identity.
   c. Give the clients strong incentives to report any key compromise incident to you. Follow the same logic as with 2.b. It would be bad policy to try to prevent key compromises by telling your clients you will charge them extra for replacing their certificates in case of an incident.

Keep this discussion in mind when you read the next chapters.

How to add a CA service to your server?

Writing a CA service for your kbmMW server is not at all technically complicated. The only thing that might make it tricky is that it has to be done so that security isn't compromised.

The first problem is that the CA service will have to have access to a CA private key. You must always take into account what will happen if the server application gets seriously compromised. One of the reasons why you should use a server certificate issued by a CA certificate instead of a self-signed server certificate, is that it is fairly unproblematic to issue just a new server certificate in case the corresponding private key gets compromised. The reason for this is that the client software only needs to know about the CA certificate, and consequently doesn't have to be replaced in case of a server key compromise.

If you extend this logic you can see why it might not be a good idea to keep both the server private key and it's CA private key in the same application: It would completely defeat the security purpose of not using a self-signed server certificate.

There are two ways to deal with this dilemma.

1. Use the same CA for both your server certificate and the client certificates and place the CA operations in a second server application running on another computer with an even higher overall security policy. This server application could be called by a service in your main application.
2. Use a different CA for the client certificates. This CA certificate might be a self-signed root certificate, provided that the client certificates are only to be used with this application.

For simplicity, this section will describe the second alternative.

Add a new data module to your application, rename it to dmCA and drop a TX509CertificateAuthority component on it. Right click on it and select Create Root Certificate. Fill in the fields and click OK. Right click on the component and save first the private key ring, then the CA certificate you created. The CA certificate must be imported into the certificate store of the TSimpleTLSInternalServer component that handles the TLS connections in your server. Right click on the latter component, select Open File, the Root Certificate File filter and open the new root certificate. You will now have two root certificates in this component: The one you used for issuing your server certificate, and the second that will be used for issuing client certificates.

Next step is to create a service that will serve the certificate requests from the clients. Create a kbmMW service with the class name TCaService, the service name 'CA_SERVICE' and a single service function called 'NEWCERT'. The automatically generated method PerformNEWCERT should look like this:

```
function TCaService.PerformNEWCERT(      ClientIdent:TkbmMWClientIdentity;
                                         const Args:array of Variant):Variant;
var
  CertReq: TCertificationRequest;
  Cert: TCertificate;
begin
  if Length(Args) <> 1 then
    kbmMWRaiseException('NEWCERT: Illegal parameter count')
  else begin
    CertReq := kbmMWVariant2Object(Args[0]) as TCertificationRequest;
    try
      if VerifyRequest(CertReq) then begin
        Cert := TCertificate.Create(nil,nil);
        try
          if not SignCertificate(CertReq,Cert) then
            kbmMWRaiseException('NEWCERT: No authorization');
          Result := kbmMWObject2Variant(Cert);
        finally
          Cert.Free;
        end;
      end else
        kbmMWRaiseException('NEWCERT: No authorization');
    finally
      CertReq.Free;
    end;
  end;
end;
```

Add the StrSecII units Pkix_Cert and Pkcs_10 to the uses clause. Add the methods VerifyRequest and SignCertificate with the following implementations:

```
function TCaService.SignCertificate(     CertReq:TCertificationRequest;
                                         DstCert:TCertificate): Boolean;
var
  Ext: TExtension;
begin
  // Extract subject:
  DstCert.Subject.CommonName := CertReq.Subject.CommonName;
  DstCert.Subject.OrganizationalUnitName :=
    CertReq.Subject.OrganizationalUnitName;
  DstCert.Subject.OrganizationName := CertReq.Subject.OrganizationName;
  DstCert.Subject.Country := CertReq.Subject.Country;
  // Extract Public Key:
  DstCert.SubjectPublicKeyInfo := CertReq.SubjectPKInfo;
  DstCert.PublicKeyIdentifier := CertReq.PublicKeyIdentifier;
  // Impose server-defined fields:
  DstCert.TbsCertificate.Validity.NotBefore.AsDateTime := Trunc(Now – OffsetFromGMT/24);
  DstCert.TbsCertificate.Validity.NotAfter.AsDateTime := Trunc(Now – OffsetFromGMT/24 + 370);
  DstCert.TbsCertificate.Extensions.KeyUsage :=
    [keyEncipherment,digitalSignature];
  Ext := CACert.TbsCertificate.Extensions.AddUniqueItem(eveIdCeExtKeyUsage);
  Ext.ExtnValue.AsCe_ExtKeyUsage.AddValue(id_kp_clientAuth);
  // Sign Certificate:
  Result := dmCA.SignCertificate(DstCert);
end;
```

```
function TCaService.VerifyRequest(CertReq:TCertificationRequest): Boolean;
var
  CommonName,
  OrganizationalUnitName,
  OrganizationName,
  Country,
  RegToken: string;
begin
  CommonName := CertReq.Subject.CommonName;
  OrganizationalUnitName := CertReq.Subject.OrganizationalUnitName;
  OrganizationName := CertReq.Subject.OrganizationName;
  Country := CertReq.Subject.Country;
  RegToken := CertReq.CertificationRequestInfo.Attributes.RegToken;
  {TODO: Add code to verify that the strings extracted above match.
   This could be a query to a customer database or whatever applies in
   your case.}
  if Result then
    Result := CertReq.CheckSignature(nil);
end;
```

The method PerformNEWCERT uses the kbmMWVariant2Object and kbmMWObject2Variant functions. You must register functions that are able to handle these conversions:

```
unit CaGlobal;

interface

uses
  Classes, kbmMWGlobal;

procedure ASNWrapperWriter(AObject:TObject; AStream:TStream);
function CertReqReader(AStream: TStream): TObject;
function CertReader(AStream: TStream): TObject;

implementation

uses
  Asn1, Pkix_Cert, Pkcs_10;

procedure ASNWrapperWriter(AObject:TObject; AStream:TStream);
begin
  (AObject as TASNCustomWrapper).SaveToStream(AStream);
end;

function CertReqReader(AStream: TStream): TObject;
begin
  Result := TCertificationRequest.Create(nil,nil);
  TCertificationRequest(Result).LoadFromStream(AStream);
end;

function CertReader(AStream: TStream): TObject;
begin
  Result := TCertificate.Create(nil,nil);
  TCertificate(Result).LoadFromStream(AStream);
end;

initialization
  kbmMWRegisterStreamableObject(TCertificationRequest,CertReqReader,
    ASNWrapperWriter);
  kbmMWRegisterStreamableObject(TCertificate,CertReader,ASNWrapperWriter);
end.
```

Add the unit CaGlobal to both your server project and your client project.

The method SignCertificate calls a method dmCA.SignCertificate you must also implement:

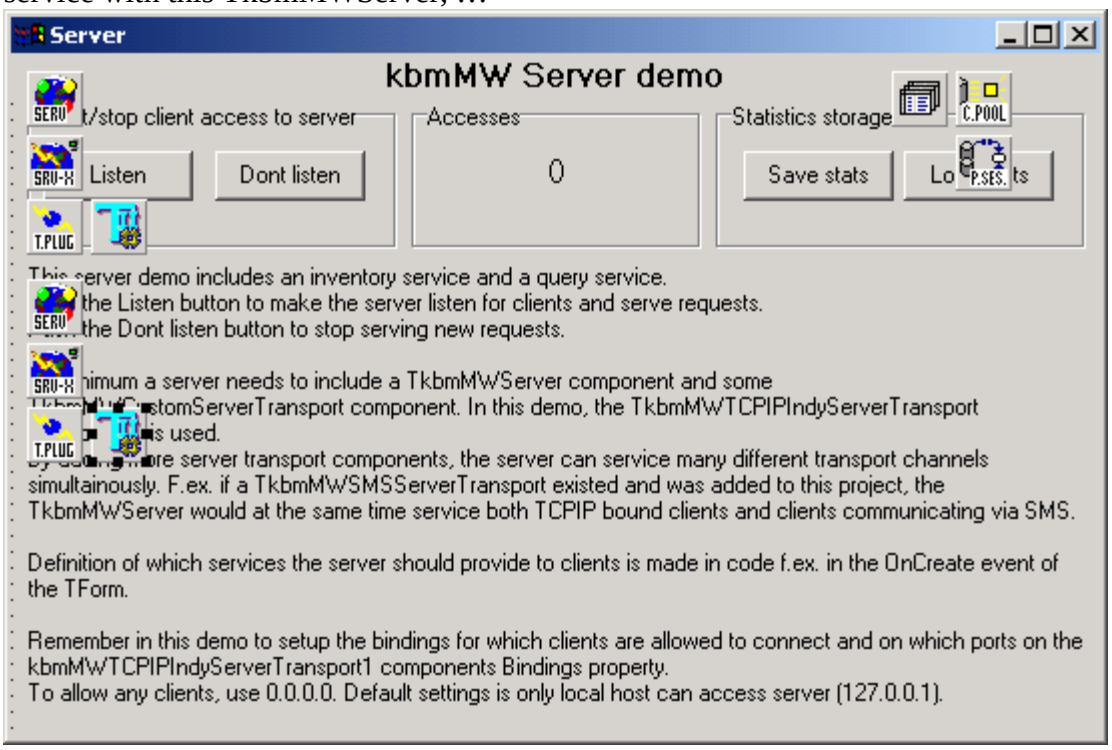
```
function TdmCA.SignCertificate(Cert: TCertificate): Boolean;
var
  Status: TCertStatusCode;
begin
  Result := X509CertificateAuthority1.SignCertificate(Cert,nil,Status);
end;
```

Remember to save the IssuedCerts store of the TX509CertificateAuthority component, as well as load it again on start up. Use the SaveIssuedCertsToSCLFile and LoadIssuedCertsFromSCLFile methods for this purpose. Failure to do so might present you with unmanageable problems for example should you ever need to revoke a certificate.

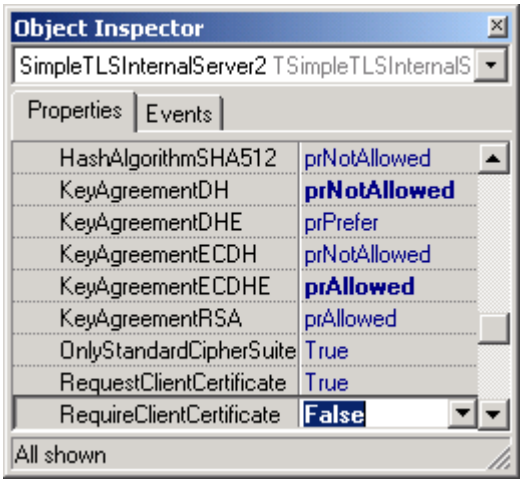
The second problem is that the purpose of the CA service is to issue the certificates that are to be used for authentication when the client requests any of the other services hosted by the server. When the client connects to send the first request to the CA service it will not have a certificate, since that is what it is out to get. Obviously, you can't require that the client authenticate itself with a certificate at this point. On the other hand, you **do** want to require the client to authenticate itself with a certificate when it connects to request any other service hosted by server.

There are three ways to accomplish this:

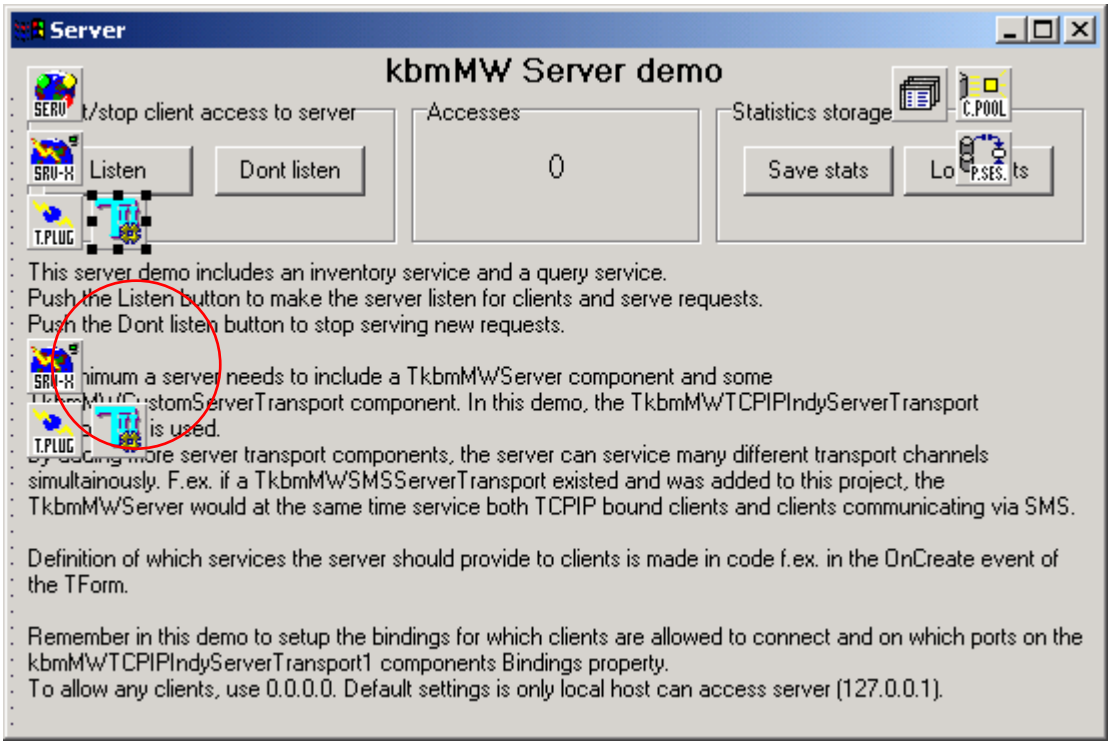
1. Add a separate chain of TSimpleTLSInternalServer, TkbmMWSsTLSTransportPlugin, TkbmMWTCPIPServerTransport and TkbmMWServer components, register only the CA service with this TkbmMWServer, …



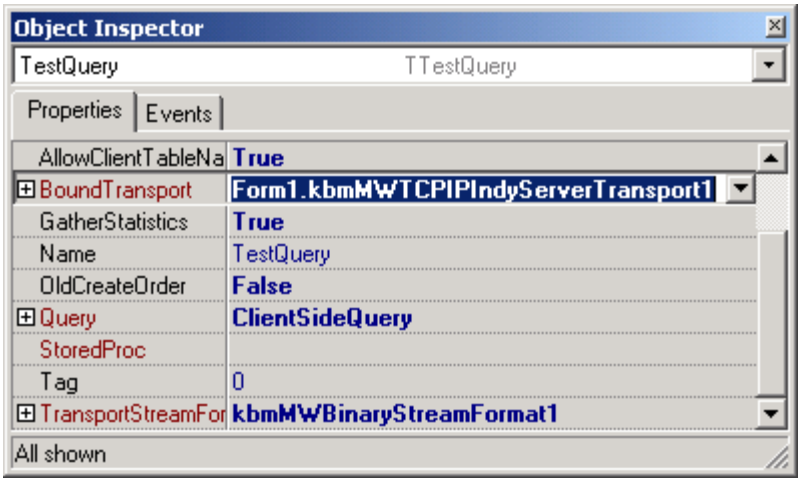
… load the same keys and certificates into both TSimpleTLSInternalServer components, and set the TSimpleTLSInternalServer.Options.RequireClientCertificate property to False of the component in the CA service chain only.



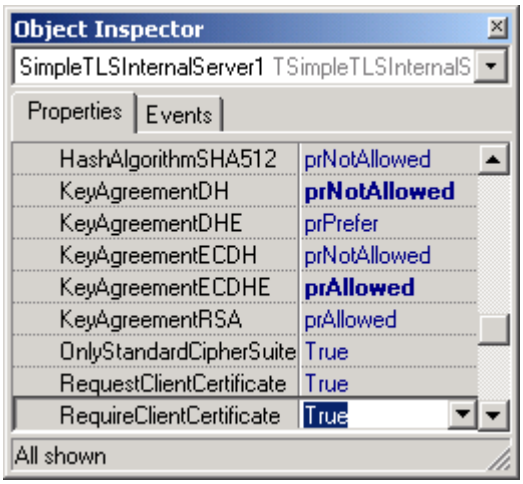
2. Add a separate chain of TSimpleTLSInternalServer, TkbmMWSsTLSTransportPlugin, and TkbmMWTCPIPServerTransport components, load the same keys and certificates into both TSimpleTLSInternalServer components, set the TSimpleTLSInternalServer.Options.RequireClientCertificate property to False of the component in this chain only,…



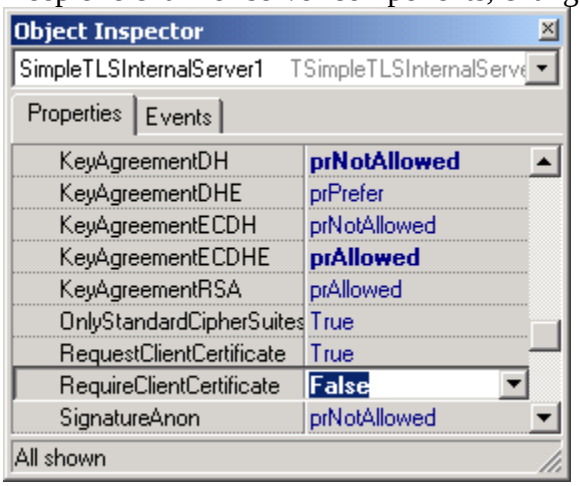
…and set the BoundTransport property of all services except the CA service to the original TkbmMWTCPIPServerTransport …



…which is linked to a TSimpleTLSInternalServer component with Options.RequireClientCertificate to True.



3. Keep one chain of server components, change Options.RequireClientCertificate to False…



…and add an OnAuthenticate event to the TkbmMWServer that looks something like this:

```
type
  TServiceHack = class(TkbmMWCustomService);

procedure TForm1.kbmMWServer1Authenticate(            Sender: TObject;
                                                      ClientIdent: TkbmMWClientIdentity;
                                                      var Perm: TkbmMWAccessPermissions);
var
  Service: TServiceHack;
  Info: TkbmMWServerTransportInfo;
begin
  if not (Sender is TcaService) then begin
    Service := TServiceHack(Sender as TkbmMWCustomService);
    Info := Service.RequestTransportStream.Info as TkbmMWServerTransportInfo;
    if (Info.Certificate as TkbmMWSsTLSSocket).GetClientCert = nil then
      Perm := [];
  end;
end;
```

Note that depending on which solution you select for the server, you will have to adjust the client in the corresponding way. Namely, add a CAClient component (e.g. TkbmMWSimpleClient) to the form, and if you select alternative 1 or 2 above you must add corresponding client transport components too. You do however not need to add separate TSimpleTLSInternalServer and TkbmMWSsTLSTransportPlugin components to the client, but might use the existing components.

All that should be left is to add a method to the client that will call the NEWCERT service function.

Firstly, the client software must obtain the name, organization, organizational unit, country and registration token from the client user. One way to obtain these values is (of course) to add a dialog with TEdit controls for each of these values. The details are left to the reader. Secondly, the client software must create the key pair the client will use for authentication. Thirdly, the certificate request must be sent to the server. Fourthly, the private key ring and the certificate has to be saved.

Add the StrSecII units Pkix, SecUtils, SsRijndael, Pkcs_10, Pkix_Cert and StreamSecII to the uses clause. The RequestCertificate method might look something like this:

```
procedure TForm1.RequestCertificate(     CommonName,OrganizationName,
                                         OrganizationalUnitName,Country:string;
                                         Password: ISecretKey);
var
  P10: TCertificationRequest;
  vCR, vCer: Variant;
  Cert: Tcertificate;
  Status: TCertStatusCode;
begin
  P10 := TCertificationRequest.Create(nil,nil);
  try
    P10.Data.ReadOnly := False;

    // Subject
    P10.Subject.CommonName := CommonName;
    P10.Subject.Country := Country;
    P10.Subject.OrganizationName := OrganizationName;
    P10.Subject.OrganizationalUnitName := OrganizationalUnitName;
    // Public Key
    with TStreamSecII.Create(nil) do try
      if not CreateKeyPair(P10,rsaEncryption,1024) then
        kbmMWRaiseException('RequestCertificate: Unable to create key');
      if not SignSigned(P10,P10,haSHA1) then
        kbmMWRaiseException('RequestCertificate: Unable to sign request');
      if not SavePrivateKeyRingToFile('User.pkr',
                                      Password,
                                      kdfWPv2SHA1,1 shl 13,
                                      id_aes256_wrap,
                                      TRijndael_ABC) then
        kbmMWRaiseException('RequestCertificate: Unable save private key ring');
    finally
      Free;
    end;
    SimpleTLSInternalServer1.LoadPrivateKeyRing('User.pkr');
    vCR := kbmMWObject2Variant(P10);
    vCer := CaClient.Request('CA_SERVICE','1.0','NEWCERT',[vCR]);
    Cert := kbmMWVariant2Object(vCer) as TCertificate;
    try
      SimpleTLSInternalServer1.AddCertificate(Cert.Data,True,Status);
      if Status <> crcOK then
        kbmMWRaiseException('Illegal certificate returned from host');
      SimpleTLSInternalServer1.SaveMyCertsToFile('User.scl');
    finally
      Cert.Free;
    end;
  finally
    P10.Free;
  end;
end;
```

Don’t forget to let the client application load the files User.pkr and User.scl, for example in the OnCreate event of the main form.

Additional Resources

Links to documents which explains different parts of SSL and digital signatures in a more down to earth language:

"Public-key encryption for dummies"
http://www.nwfusion.com/news/64452_05-17-1999.html

"How do digital signatures work?"
http://www.howstuffworks.com/question571.htm

"How Encryption and Digital Signatures Work"
http://www.tatanka.com/doc/technote/tn0035.htm

"Security in One-Line Governance"
http://www.cc.ioc.ee/training/unesco/onlinegov/security/

Links to much more hardcore technical documents
http://www.streamsec.com/links.asp

Kim Madsen
Components4Developers