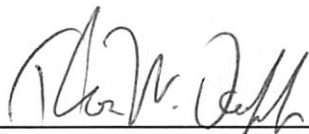


# Security in Reliable Multicasting

(Security for the Electronic Notebook Project)

Qiang Ye  
Department of Computer Science  
Brown University  
[qy@cs.brown.edu](mailto:qy@cs.brown.edu)  
May 2003

Submitted in partial fulfillment of the requirements for the Degree of Master of Science  
in the Department of Computer Science at Brown University



---

Signature (Professor Thomas W. Doepner Jr.)

5/16/03

Date

# Security in Reliable Multicasting

(Security for the Electronic Notebook Project)

Qiang Ye

Department of Computer Science

Brown University

[qy@cs.brown.edu](mailto:qy@cs.brown.edu)

May 2003

## 1 Introduction<sup>1</sup>

Consider the old-fashioned paper notebook. Everyone is familiar with it and needs no instruction on its use. We go to class or a meeting and use it to take notes. We can annotate any printed matter that is handed out and stuff it in the notebook. Later we can go over our notes and handouts, further annotate them, and perhaps share them with others (with the help of a copying machine).

However, despite amazing progress in computer technology, we really have no useful electronic analog of the notebook. We can certainly take notes on a computer; we can receive documents on a computer; we can annotate documents stored on a computer and send them to others on a computer. But none of this adds up to the convenience of a notebook.

What's missing? One obvious thing is the ergonomics of the hardware. Taking notes or annotating a document using a laptop computer is not as painless as doing it on a pad of paper contained in the notebook. Fortunately, with some limitation, this issue is being addressed with the newly availability of Tablet PC computers, wireless networks and Microsoft Windows Tablet PC Edition. But this deals only with note taking and communication. It will help a lot, but it's still not a notebook. What else is missing?

Notebooks are easy to use and are universal. The only compatibility issue we have to worry about is paper size. Distributing handouts (though not producing them) is uncomplicated. Everything can be annotated using the one and only annotation interface. Producing copies for others is well understood and straightforward (though not always convenient). Notebooks are less subject to failures than are computers: what I write in them will likely survive power problems, viruses, and being dropped from moderate heights. And it is easy to authenticate the person you give your documents to.

The object of the electronic notebook project is to build an electronic analog of the paper based notebook by adding the missing part of the state of the art. An electronic notebook

---

<sup>1</sup> Part of this section is adapted from *An Electronic Notebook: A Proposal to Microsoft Research* by Professor Thomas W. Doeppner Jr. October 29, 2001.

should be able to do everything the paper version can but provides enough additional useful functionality to make paper-based notebooks go the way of typewriters.

The following are the three basic scenarios to which our project is targeted:

- 1) **The lecture.** Students are taking a lecture-oriented course. Notes for each lecture are distributed just prior to the lecture. During the lecture they can look at additional materials, such as web pages. They annotate the notes and other materials, share, and study them. They further annotate the materials afterwards, perhaps in their dorm rooms or perhaps on a train or plane while traveling. Occasionally the professor finds an error in the notes and distributes a new version. Students incorporate the new version into their annotated versions (or vice versa).
- 2) **The meeting.** Participants are given an agenda. They take private notes, possibly by annotating the agenda. During the meeting various participants distribute materials that are gone over by all. Suggestions are made and incorporated into the materials. Whoever is responsible for official minutes distributes them shortly after the meeting. Corrections are made and the minutes become the official record. Participants incorporate their own and others' annotations into their copies of the minutes. In this scenario, any materials that are distributed during the meeting should be kept within the participants. The communication of the meeting should be protected from security breach introduced by the un-secure nature of wireless network.
- 3) **The museum trip.** We tour the museum. Many of the exhibit areas have write-ups describing what's on display, perhaps with photographs. Individual exhibits have labels. We collect or copy these as we walk through and jot down our thoughts on what we see. Later we share our notes with one another and perhaps incorporate them into articles or research papers we're writing about the museum or the subject area.

This paper deals with all the security related issues in the electronic notebook project as a whole. It consists of three main parts:

- (1) Mutual authentication and session key management (its generation, storage and distribution);
- (2) Secure communication during normal operating and
- (3) Multicast group leadership handover.

It can be seen that different application scenarios require different levels of security. The meeting scenario described above should have higher security requirement than either of the other two. Nonetheless, the security work should be built on a common framework. Some of the security components can be set as optional or user configurable and the system administrator (in the project we call it the group leader) can choose to turn them on and off according to the security requirement of the application, or through profiling.

## 2 Design Goals

In this section, we discuss the several security goals that we want to achieve in the electronic notebook project.

### ***2.1 Mutual authentication and session key management***

In the three application scenarios discussed above in section 1, a multicasting group for an application is usually formed in an ad-hoc fashion. Members will be able to join and leave at any time. The formation and routing processes are out of the scope of this paper. However, when a person requests to join a group, he/she needs to do a mutual authentication with the group before the real group joining protocol begins (or as the first step of the group joining process). That is, the group (through its leader) needs to verify that the person is indeed the one who claims to be. Likewise, the person can verify about the group.

The only session key used for the whole group, which is managed and distributed by the group leader, is then distributed to the person who will then use the session key as a proof when joining the group.

Public key based (using security certificate) authentication similar to what is used in SSL protocol is a good candidate here. If we want to use this approach, we need to require that everyone who wants to join the group holds a verifiable certificate. By saying verifiable, I mean that either the certificate has a chain that ends in a trusted root CA (operating system build-in trusted root CA) if we are connected to the internet, or the certificates are signed by an organization whose certificate everyone has cached in his system. For example, all certificates are signed by Brown University.

Unlike a secure web site using SSL, whose domain name we can check with DNS entry, and compare it with the CN in the certificate, no such a directory exists for the time being against which we can check the person's name and determine if it matches the CN in his certificate. Hence, when verifying the certificates, we might need to ignore the CN part as we usually do with SSL.

The authentication requirement should apply to all of the known application scenarios. However, the requirement for each scenario should not be the same. For example, in a business meeting, we might have a list of all the attendances and check the names against the CN part in the certificate. In other occasions, this step could be omitted.

### ***2.2 Data integrity check (Tamper detection)***

Everyone in a group is able to send data to the group. Although we assume that there will be a leader in each group (see the next section), the duty for the leader is limited to mutual authentication (verifying the identity of new members) and managing the session key. The leader should not be responsible for filtering every data item and verify its source identity. Instead, group members who are interested in the data item should be able to verify the origin and detect any tamper.

To achieve the highest integrity check requirement, we should require that every piece of data be signed by the sender with his/her private key. However, asymmetric approach is much more expensive than symmetric one. And due to the length of keys, in this project



we should allow the user to select an option to use private key or session key to sign the data, depending on the application scenario. This could be done through application profiling.

If we only need to make sure that the document we received is from one of the members in the group and has not been tampered by anyone outside the group, signing the document with the common session is enough. If we want to be able to verify the origin of a document (i.e. it is coming from a particular member) in addition to the above requirement, the document in question needs to be signed by the owner's private key.

### **2.3 Non-repudiation (optional)**

In some situations, we may want to enforce the non-repudiation requirement, that is, to prevent the sender of data item from claiming at a later time that the item was never sent. Such a requirement would be rare in our targeted application scenarios, though. If this is indeed required, it is easily implemented together with the data integrity check part. Of course this has to be accomplished with signing every document with its owner's private key, which is quite expensive.

### **2.4 Confidentiality (optional)**

Again, in some situations, such as a business meeting, we might want to send and receive sensitive information between group members. To fulfill such a requirement, communication between group members must be encrypted and decrypted. However, in the lecture and museum scenarios, the need for data encryption is not high. In such a case the encryption and decryption overhead should be avoided. Hence this part is optional. In the case that confidentiality is a requirement, the common session key could be used for encryption and decryption.

### **2.5 Least user interference**

As the target users of the application produced by this project are mostly non-technical persons, e.g. humanities faculties, business men and museum tourists, we want the security implementation of this project be as user-transparent as possible. In the normal cases, the users should not worry about dealing with the security issues; instead, the system should do the most of the job. Only when there is something abnormal happening, e.g. attacks, malicious users, malicious groups, should the users be warned. And even in such a case, the system should do most of the job.

The security protocol should stay in the backend and should act as transparently as possible to the user. The user should not be bothered if all goes well, and only be notified if serious security problem has happened. I believe this should be the basic approach for all applications that need security in place.

### **2.6 System robustness (Group leadership handover)**

Last but not least, the security protocol should be robust. This could be the harder part of the project.

We assume that there is a leader for each group who is in charge of accepting new members and managing the session keys. Obviously, this is true for most cases.

When someone requests to join the group, the join request is routed to the group leader. The following handshaking protocol happens between the one who requests to join the group and the group leader through uni-cast communication, until the new-comer securely and officially joins the group.

Since the group leader is responsible for authenticating with new members and managing the session key, this could become a single point failure. Although the communication between existing group members will not be affected should the leader disappear for any reason, new members will no longer be able to join the group. For the sake of the robustness of the system, we must have means to deal with this kind of situation.

There are two different kinds of handovers, voluntary and involuntary. The first one is the easier one, in which the leader is scheduled to leave and could arrange the handover of leadership, including the handover of the directory entries cache.

The other one refers to the situation in which the leader's computer crashes. This is difficult to deal with. One possible solution would be the leader periodically multicasts to the group a keep-alive packet. If this packet is not heard by group members for a pre-determined time, the leader is assumed to have disappeared and the group needs to decide a new leader using some election mechanism.

## **2.7 Some assumptions**

This section defines some assumptions we will make in the development of the project.

### **2.7.1 Everyone has a security certificate**

We want to use the public key infrastructure (PKI) approach to do peer authentication and session key distribution. Public key approach requires that each peer who wants to be authenticated has a security certificate installed in his computer. In our project, we want to be able to do two way authentication, meaning that one who wants to join a group needs to make sure that the group is indeed the group that it claims to be, and on the other hand the group (its leader) needs to make sure that the person who requests to join the group is indeed the one whom he claims to be.

To use this approach in our project, we assume that each participant has a security certificate. Hence the authentication could be done through members' security certificates.

Furthermore, for group members that are not the group leader, they only need a client authentication certificate. For the group leader, he must have a server authentication certificate.

### **2.7.2 Certificates could be verified**

To be able to do the authentication, we further assume that certificates can be verified due to SSL protocol requirements, i.e. the Certificate Authority (CA) or CAs who issued the certificates, and who either can be reached when needed or is built into the system.

(One approach to fulfill this requirement is to assume that all the potential members have certificates that are issued by a common organization that the system can trust, such as assuming we all have a certificate issued by Brown University.)

We also assume that there exists a directory server that supports Lightweight Directory Access Protocol (LDAP) or some other directory services so that we can check every certificate against entries in the directory. This is a heavy weight approach in that it requires that we have network connections all the time.

Since such a directory server is not always available in reality, especially when in ad-hoc mode, the step of checking certificate against a directory could be made optional in most scenarios. However, in situations that this is required, such as in some business meetings, the group leader (meeting organizer) could access the directory server beforehand for entries for all the allowed persons in the meeting and cache them in local host, and later check certificates against this cache.

Alternatively, for the common case in which all group members are in the same organization, everyone's certificate could be signed by a CA, whose public key is held by all. Thus all members can verify each other's certificate.

Another possible approach is to exchange the certificate via the IrDA port, which exists on almost all laptops nowadays. Unfortunately, it seems that some newer Tablet PC has dropped the support for the IrDA port since this technology has never taken off in reality.

### **2.7.3 Multicast group addresses are well know**

The name and address of a multicast group that one would like to join is published so that one can select the group to join by a simple click on one of the names in the UI without having to manually input the group address or name. This is not an issue that is directly related to security. However, during developing period, we need to make the UI part of the security project.

### **3 Protocol design**

As mentioned earlier, we choose to use public key infrastructure (PKI) approach to do the authentication. At first we tried to design our own authentication handshake protocol similar to the SSL protocol. (The protocol we put together is attached to this report as an appendix.) Later we realized that it is kind of dangerous to design a brand new security protocol by our own in the sense that it might introduce some “holes”.

So we now use instead the actual SSL protocol with the client authentication option enabled (mutual authentication) to do the two way authentication and through the established secure SSL channel to distribute the session key. The new member, upon receiving the session key, then begin the next step in the multicast group joining process with the acquired session key, since knowing the session means the person has authenticated to the group through the leader. We could decide how to use the session key in the joining process. Two approaches are available: one is to encrypt the join message with the session key and the other is to sign it.

#### **3.1 Mutual (Two-way) Authentication**

We choose to use the proven SSL protocol (with client authentication option enabled) to do the two-way authentication between the new-comer and the group leader using unicast communication through a well-known port number. If anything went wrong, the underlying SSL protocol will report errors and information about the error so we can handle in the program or report to the user. Otherwise, if SSL finishes the authentication without any problem, the group leader could use the established secure channel to distribute the session key to the new comer. The SSL channel is then closed and the new comer could use the session key to do the actual multicast group joining process, either by encrypting all the messages or just signing them. We will evaluate these two options later.

#### **3.2 Session Key**

The session key, together with the initial vector, is generated in a cryptographically secure manner when the group leader starts the session. Or it can also be manually assigned to a pre-selected one by the group leader.

Once the session key is generated, it is used throughout the whole multicast group session and is managed by the group leader.

#### **3.3 Communications**

After the handshake is successfully finished and the group formed, all following communications between group members could use the same and only session key, together with the initial vector, to fulfill the security requirement.

##### **3.3.1 Data integrity (Tamper detection)**

Data integrity check is the only required part in the communication for all application scenarios. This is done with hashing algorithm.

We compute the hash code for any data item by appending the session key and initial vector to the end of the original data. The original data together with the hash code is sent, but not the session key and the initial vector.

All parties in a group are required to hash data items in this way before sending the message out, so that other members in the group can check their integrity when they receive the data.

The session key is the common secret among the group members and no one else can acquire it based on the handshake procedure. Hence if we can verify the hash of a data item with the session key, we are guaranteed that it is from inside the group and not tampered in any form. In other words, if the data item is altered in transition by an outsider, the alternation can be detected.

However, this approach does not prevent tampering from inside, i.e., some members in the group alternate the content of the data sent by other members. If we want to achieve this degree of data integrity guarantee, we must require group members to sign the data items they send with their own private key in their certificates.

### **3.3.2 Non-repudiation (optional)**

This is optional, and can be achieved easily if we take the second approach in data integrity checking, i.e., each member uses his own private key to sign the data item he sent. Since the private key in one's certificate is one's own secret, he/she can't deny at a later time that he/she hasn't sent the data item.

If we use the common session key to sign data items, then the requirement is hard to fulfill.

### **3.3.3 Encryption (optional)**

In some situations, which are especially true for some business meetings, we want to keep the content of the whole meeting confidential. In such an application, we could encrypt any data item using the session key and decrypt it when receiving. Again based on the handshake procedure and distribute mechanism of the session key, the session is only known by group members, and it is a common secret between all group members. We are guaranteed that the information in the session is kept confidential.

## 4. Security Certificates

Before jumping into the actual implementation, I would like to first discuss in this section the way how to get certificates to be used in this project, since without usable security certificates in place, the implementation code will make no sense.

### 4.1 How to get a free verifiable test certificate

There are several options that you can get a free verifiable security certificate to test out the security functions in this project.

1. Best option is: Microsoft's free test Certificate Authority Server located at: <http://131.107.152.153>. Just click on "Request a certificate", "Web Browser Certificate" "Use advanced Certificate form". Fill out the fields. Generate both Client and Server Authentication Certificates. Leave the defaults except for Key size, change it to a bigger one if you want. You can also generate SSL test certificate for your IIS server there if you want.
2. The CREN web site at: <http://ca.cren.net> is offering free test certificates. The limitation of this service is that it only provides client authentication or email security certificates. It does not provide server authenticates for free, so you need to go the other ways to get one for the leader. Just click "CREN Test Certificate Authority" and follow the link and the instructions to get your free test certificate.
3. If you use option 2, you need to get a separate certificate for server authenticate certificate. If you have access to a Windows XP/2000 machine with IIS installed, you can use the IIS installation on that machine to generate a SSL server certificate request and send it to VeriSign to be signed by its test CA for free. The downside is that it is only valid for 15 days.
4. You can explore the options offered by OpenSSL. Check this FAQ for more info: <http://www.openssl.org/docs/HOWTO/certificates.txt>
5. Another option is using the built-in CA of windows 2000 Server edition. You can find step by step example in ATL7 SecureSOAP Sample's readme that comes with VS.NET MSDN.
6. Yet another option is to use makecert.exe that comes with .NET SDK/PSDK. A new version that comes with current PSDK supports exportable private keys (-pe option).

### 4.2 How to get a real world certificate

To get a real world security certificate, we can order one from [www.verisign.com](http://www.verisign.com) or [www.thawte.com](http://www.thawte.com) or some other commercial CAs. Each CA has its own rules to follow when you request/order a security certificate.



### 4.3 Where do I put my certificates

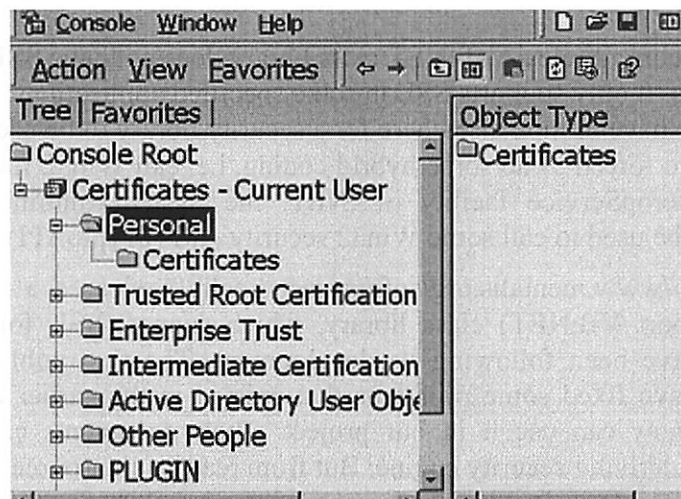
The implementation of this project supports two ways to store and use the security certificates. The first one is to install them into the System Certificate Store and the other is to store the certificates as files on disk. I recommend the first approach for simplicity manageability.

If you store your certificates in the system certificate store, your personal certificates will be stored under MY store.

Windows XP offers a high security option when you import your certificate to the MY store, which enables password protection for your private key in the certificate. If you enable this option, the system will ask you for the password every time the private key is used. This could become annoying over time. Furthermore, if you enable this option for the server certificate, you will have trouble loading the certificate to be used by the server. Hence I recommend not enabling this option.

To View your current certificates on Windows XP/2000/NT: Run "mmc.exe". Go to Console | Add/Remove Snap-in | Add | Certificates.

Your personal or "MY" store will be under Personal | Certificates tree items.



If you prefer to store the certificate in a disk file, and let the code load it from that file, you can do so. The implementation supports two certificate file format, DER Encoded Binary X.509 and PKCS#7 Certificate, the first one has a .cer extension name and the second one has a .p7b or .pfx extension name and it supports password protection.

## 5 Implementation

### 5.1 Target platform

The target running platform is Windows XP TabletPC Edition with Microsoft .NET framework support. In particular, we are using the C# programming language and utilizing the CLR (Common Language Runtime) library introduced in the .NET framework.

On the other hand, this implementation is not limited to the Windows XP TabletPC Edition. The result code is also tested successfully on Windows XP professional version and Windows 2000 professional version.

The result code is a class library that can be called from the reliable multicast code and in the form of a DLL file. Two test applications using the class library is also included.

Microsoft has PKI built into Windows 2000 and Windows XP through an API called CryptoAPI, which is an interface for all security related Win32 applications. And it implemented a basic security service provider in the system. We can also install our own security service provider to the system. The Microsoft implementation of SSL support is called SChannel.

Unfortunately, the current release of .NET class library does not have SSL support for the general socket class. It only supports SSL in some specified applications. That is not what we need. Hence we have to look elsewhere for a security library to be used in our project. Or otherwise we are forced to do some hybrid coding, i.e. call Win32 functions from our C# code. The InteropService facility of .NET, the System.Runtime.InteropServices Namespace, could be used to call some Win32 security API (CryptoAPI) if we have to.

Mentalis.org (<http://www.mentalis.org/soft/projects/secplib/>) released a free open source C# (and also support VB.NET) class library, which support SSL for general socket application. We have been following its development. The first public release of the library seems to have fixed some problems that existed in the earlier developer's beta releases, and we now can use it in our project. Their documents claim that it uses OpenSSL as the underlying security engine. But from reading the source code, it seems to me they just used the built-in Microsoft implementation, SChannel, as the underlying engine. But since CryptoAPI is just an interface, we can install OpenSSL to the system and selection that as the underlying security service provider if we want to.

In the implementation, for the sake of convenience, we call the group leader the "server" and other group members need to authenticate to the leader "client", since their respective roles fall into the server-client definitions.

### 5.2 Code Organization

The main code is organized in a hierarchy fashion. At the top it is an abstract class called Authenticator, which represents the base class for the security entity in the system for the electronic notebook from which both the secServer and secClient classes must derive. It has some methods that are common to both the subclasses. The secServer class is the code that is running on the group leader's system, which represents the group and does mutual authenticate with new group members and takes care of the group session key.

The secClient class is the code that is running on all group members' systems, which does mutual authentication with the group leader and retrieve the group session key.

There are also some helper classes that provide support for the main code.

### 5.3 The Authenticator base class

The Authenticator base class includes some methods that are common to both server and client subclasses.

The main methods are:

```
public ICryptoTransform CreateDecryptor()  
public ICryptoTransform CreateEncryptor()
```

These two methods each returns an ICryptoTransform object with the group session key and initial vector with which the user can create a CryptoStream to encrypt/decrypt stream data as used in the following example.

```
Socket sock = new Socket( AddressFamily.InterNetwork,  
                           SocketType.Stream, ProtocolType.Tcp );  
sock.Connect( new IPEndPoint( Dns.Resolve( host ).AddressList[0], port ) );  
NetworkStream strm = new NetworkStream( sock );  
CryptoStream cstrm = new CryptoStream( strm, client.CreateEncryptor(),  
                                       CryptoStreamMode.Write );  
byte[] buff = Encoding.ASCII.GetBytes( StringMessageToBeSend );  
cstrm.Write( buff, 0, buff.Length );  
cstrm.Flush();  
cstrm.Close();  
strm.Close();  
sock.Close();
```

Usually you use the Encryptor to create a stream to send messages, and use the Decryptor to receive messages.

```
public byte[] ComputeHash( byte[] message )
```

The user calls this method on some messages to compute the hash code. The implementation internally uses SHA1 algorithm for the computation. The hash code is computed by appending the session key and initial vector to the end of the original message. This is used for message integrity check. The session key and initial vector are not transmitted with the message; hence if someone from outside of the group tries to alter the message, it can be detected. It can also detect transmission errors.

Other members of this class include storage for the session key and initial vector and properties for the local certificate and local network endpoint in use.

Properties about the local certificate:

```
public X509Certificate LocalCertificate  
public string LocalCertString
```

There are two read only properties. The first gets back the certificate in X.509 format, which is a .NET built in type. The other gets back the string representation of it.

Properties about the local network interface:

```
public Endpoint LocalEndPoint
```

This is a read write property, with which the user can read back the local endpoint or change it to a different one from what was set when the object was constructed.

Properties about the session key and the initial vector:

```
public byte[] SessionKey
public byte[] InitialVector
public string SessionKeyString
public string InitialVectorString
```

The first two are read write ones. They are used to read back the session key and initial vector in byte array format or set them to a different value. The later two are read only and are used to get back the string representation.

## 5.4 The *secServer* subclass

The *secServer* class is a subclass of the *Authenticator*. It has 6 overloaded constructors.

```
public secServer()
public secServer( string certfile )
public secServer( string certfile, string password )
public secServer( int port )
public secServer( int port, string certfile )
public secServer( int port, string certfile, string password )
```

The first three do not provide a port number. The class will use the default port number 50000. If the “certfile” name is not given in the constructor, the class will automatically search the system “MY” or “personal” certificate store for a qualified server authenticate certificate to load. A qualified server authenticate certificate must have an object identification (OID) of “1.3.6.1.5.5.7.3.1” and the user has the private key for it. If a “certfile” name is given without the password, then it is assumed that the certificate file is a DER format file, which can be opened without a password; otherwise it is PKCS#7 format certificate file. The constructor will throw a *SecurityException* if no qualified certificate could be loaded.

After the server object is created, the following methods should be called to start or stop the authenticate server.

```
public virtual void Start()
public virtual void Stop()
```

The secServer runs on the group leader's machine and listens on a dedicated port. It uses SSL protocol with mutual authentication option turned on to verify the incoming connection from clients, or group members, for mutual authentication.

The session key and initial vector are generated in a cryptographically secure way when the Start method is called and destroyed when the Stop method is called. The user can use the SessionKey and InitialVector properties, which are inherited from the base class, to read them back or to manually set them to a different set. If the manual setting of the session key happens before the Start method is called, this setting will be used instead of the system generated random one.

For every incoming connection from the client, the authentication happens in the underlying SSL protocol layer. If all goes well, the server sends over the session key and the initial vector to the client using the established secure connection. The client, armed with the session key and the initial vector, can then join the multicast group and communicate with other members in a secure way.

If so chosen, the server could also send additional application defined data to the client to be used in the session.

If there is any problem during the SSL authentication process, it throws exception to alert the up level application.

A regular server certificate usually includes its DNS name as the distinguish name in it and it could be verified through comparing the included domain name and the result of a DNS query. In our case, however, we are not expecting that every participant has a valid domain name for the computer he/she uses and further more to bind it to his/her security certificate. Hence, we choose to ignore the common name verification part and accept the given name. And in some situations, when we have a list of allowed participants, we can compare the name in the certificates we received to that of the list.

The one SSL connection with this client is then terminated.

The server keeps a list of the clients that have been authenticated to the group and their certificates. There is another list that keeps the information of the clients that failed the authentication. Both lists could be retrieved or displayed using the following methods and properties.

```
public void ShowClientsInfo()  
public int NumOfAuthenticated  
public int NumOfDenied  
public GroupMember[] Authenticated  
public GroupMember[] Denied
```

## **5.5 The secClient subclass**

Similar to secServer class, this secClient class is also a subclass of Authenticator. It has 6 overloaded constructors.

```

    public secClient( string host )
    public secClient( string host, string certfile )
    public secClient( string host, string certfile, string password )
    public secClient( string host, int port )
    public secClient( string host, int port, string certfile )
    public secClient( string host, int port, string certfile, string
password )

```

The parameter “host” is the host name or IP address of the machine that the authenticate server is running. The parameter “port” is the port number to use, or if it is not provided, the system default, 50000, is used. The rest of the parameters tell the class how to load the client authenticate certificate to load in a similar way as that of the secServer. The OID for a client authenticate certificate is “1.3.6.1.5.5.7.3.2”.

If a qualified certificate can’t be loaded, a SecurityException is thrown.

Then the system connects to the secServer and does SSL mutual authentication with the server.

After the successful mutual authentication, the client receives the session key and the initial vector from the server through the established secure channel, and caches them in the client object. The property

```

    public bool Authenticated

```

is set and can be checked.

This SSL connection is then terminated.

Armed with the session key and initial vector, the methods and properties inherited from the base class Authenticator can be used by up-level application to communicate with other group members securely.

This class also provides some additional properties:

```

    public X509Certificate RemoteCertificate
    public string RemoteCertString
    public EndPoint RemoteEndPoint

```

to let the user query information about the remote authentication server.

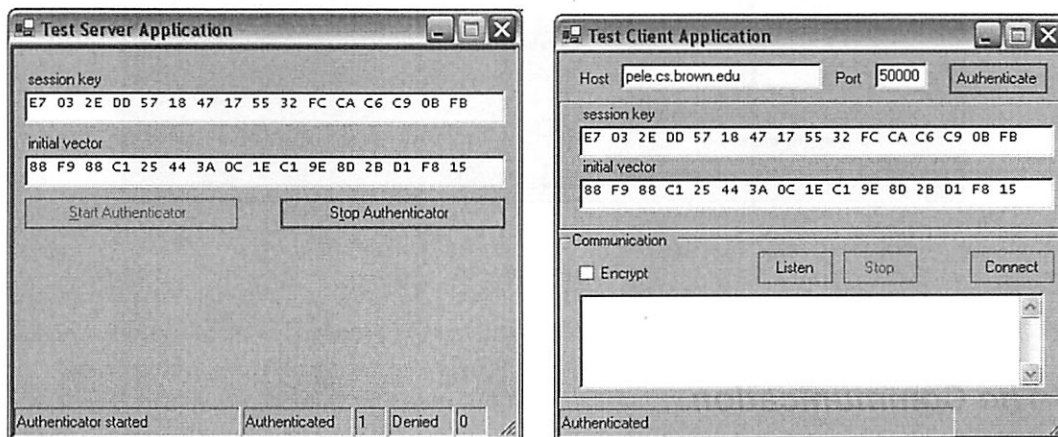


## 6 Some Test Results

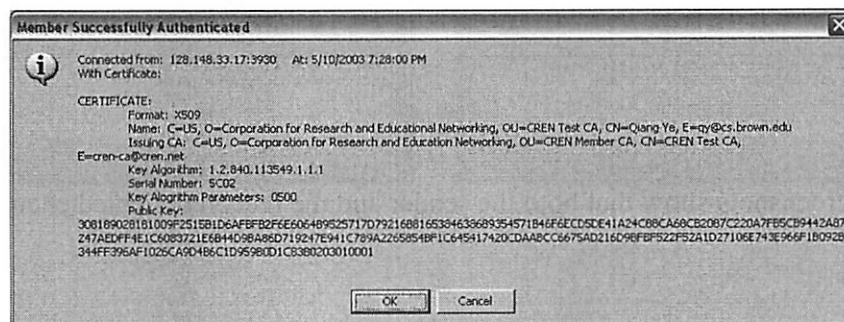
This section should discuss the results we get from the implementation.

### 6.1 The interfaces

There are two test applications coded to test out the security library implementation. One is called TestServerApp, which is to test the secServer class, and the other is called TestClientApp, which is to test secClient class. Below are the interfaces for these two test applications.

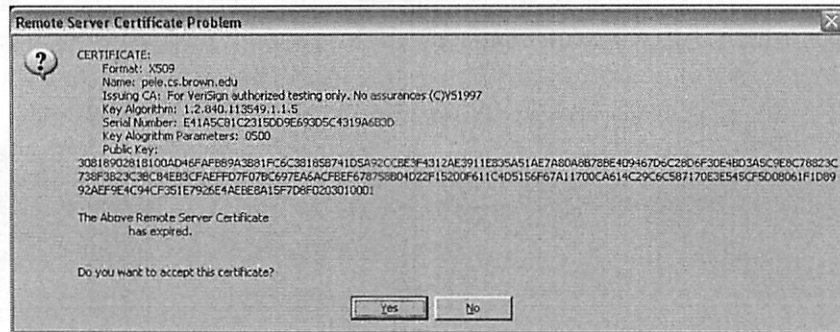


The status bar on the bottom of the Test Server Application Frame shows its current status and the numbers of clients (members) who have been authenticated and those who have been denied. Click on it, and you can read the information about the clients as shown on the screen shot below.



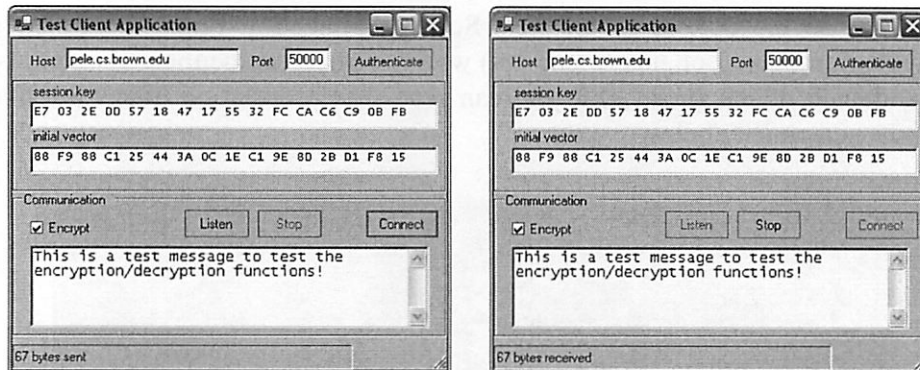
## 6.2 The Error Message in Certificate Validation

The screen shot below shows the message box that pops up when the remote certificate has a problem in the process of authentication. It asks the user to manually check the status of the certificate and decide whether to accept it. The screen shot shows that the certificate the system received has expired. This is because I have a test certificate installed which was only valid for 15 days and has expired.

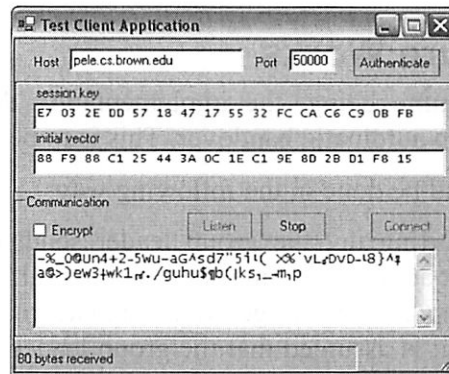
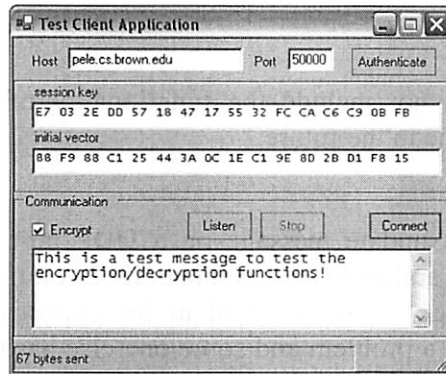


## 6.3 The Communication

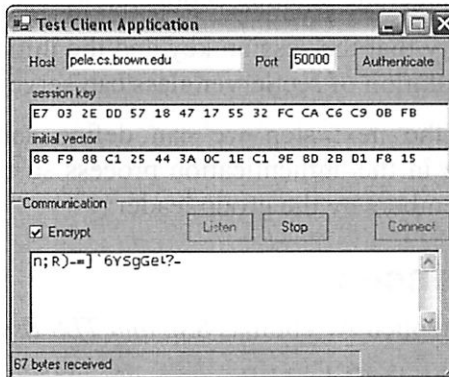
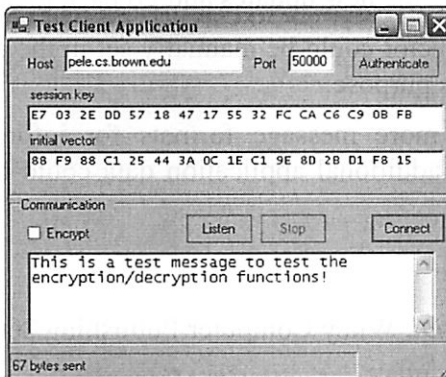
The following screen shots show the communications between two authenticated clients. On the left side are the sender's, and the right side are the receiver's.



These two screenshots show that both the sender and the receiver enabled encryption.



These two screenshots show that the sender enabled encryption, but the receiver didn't, hence it can't see the original message.



These two screenshots show that the sender didn't enable encryption and the receiver enabled it. The receiver gets the correct number of bytes, but it was not the readable message.

## 7 Future Works

For the time being, the implementation does not include the functionality of group leadership automatic handover. This is to be done in the future.

This could be done in the following way:

When starting up, the group leader sends out update message on a UDP port to the broadcast address in some set time interval. All authenticated members listen on that port for the updated message. If an update message is not received in the expected time interval, it is assumed that the group leader has a problem and some one else has to take over the leadership. Every client should wait for a random time period before trying to do so. The first client who takes over the leadership will then broadcast the updated message. Others cancel the pending actions upon receiving the new updates and go back to normal.

The ability that the session key and the initial vector could be manually set in the current implementation of secServer class can serve this purpose.

Also in the next step we can define some more message formats for information exchange in the authentication process so that additional application data could be sent out to members by the group leader.

## References

1. Stephen A. Thomas *SSL and TLS Essential*, Wiley Computer Publishing 2000
2. Charles Petzold *Programming Microsoft Windows with C#*, Microsoft Press 2002
3. Richard Bondi *Cryptography for Visual Basic, A Programmer's Guide to the Microsoft CryptoAPI*, Wiley Computer Publishing 1999
4. David S. Platt *Introducing Microsoft .NET*, Microsoft Press 2001
5. Tom Archer & Andrew Whitechapel *Inside C#*, Microsoft Press 2002
6. Microsoft *Microsoft C# Language Specifications*, Microsoft Press 2001
7. Jeffrey Richter *Applied Microsoft .NET Framework Programming*, Microsoft Press 2001
8. Mentalis.org *.NET Security Library*, <http://www.mentalis.org/soft/projects/seclib/>
9. IETF *RFC2459 Internet X.509 Public Key Infrastructure Certificate and CRL Profile*

# Appendix A

## *Our Original Authentication Protocol*

This appendix is included to show our original authentication protocol design and may include some potential security ‘holes’ of it.

I think it is worth leaving this protocol here as an appendix as we have put so much effort in its development.

### **A.1 Handshake**

The handshaking process here is similar to that of SSL. However due to the different application environment of our project from that of SSL, we have some differences from the SSL protocol.

Our protocol uses a combination of public key and symmetric key encryption. Symmetric key encryption is much faster than public key encryption. On the other hand, public key approach provides easier key management techniques.

An application session always begins with an exchange of messages called the **handshake**. The handshake allows a new member who wants to join a group to authenticate himself to the group (through the group leader) using public key techniques. After verifying the new member’s identity, the group (through its leader) then authenticates itself to the new member also using public key technique. And at the same time the group securely distributes the session key to the authorized new member. The session key is then used for rapid encryption, decryption, and tamper detection during the session that follows.

The session key for an application session is generated when the first member besides the group leader joins the group. Then the same session key is used for the whole session and shared by all authorized group members.

The following are the steps involved in the handshake. (Handshake is done through uni-cast between the new member and the group leader. For the simplicity of explanation, in the following text, server refers to the group (through its leader) and client refers to a new member who requests to join the group. But note that they are not really server and client.)

(Another issue related to this part is the communicate port number set up and discovery process. For the sake of simplicity, we could assign a fix number above the system port number range for now. Ultimately, we need a way to let the group select a port number at runtime and advertise it and a new member could find it through some mechanism. This part need to be further investigated.)

1. The client sends to the server a join group request along with it, he also sends some randomly generated data, the signature of the random data using its own private key, and its certificate, and maybe some other information needed for the communication between the server and the client.
2. The server uses some of the information sent by the client to authenticate the client (see section 3 for details). If the client cannot be authenticated, the group leader is warned of the problem and informed that some one is trying to join the group but his identity cannot be authenticated. If the client can be successfully authenticated, the server goes on to Step 3.
3. Optionally, the server can check that the client's certificate is present in the client's entry in an LDAP directory. This optional step provides one way of ensuring that the client's certificate has not been revoked. In the Ad-Hoc mode, this directory entry should be fetched from the directory server beforehand and stored in the local cache.
4. If this client is the first one who requested to join the group. The server also generates some random data and uses this data together with the random data sent by the client to generate a

session key to be used in the following application session. Otherwise the server just uses the existing session key.

5. The server first encrypts the session key with its own private key, then again encrypts the encrypted key with the client's public key (obtained from the client's certificate, sent in Step 1, and verified in Step 3), and sends the double encrypted session key to the client.
6. The server also signs another piece of randomly generated data. The server sends both the signed data and the server's own certificate to the client along with the double encrypted session key.
7. The client, upon receipt of the packet sent by the server in step 6, attempts to authenticate the server (see section 4 for details). If the server cannot be authenticated, the session is terminated. If the server can be successfully authenticated, go to step 8.
8. Optionally, the client can check that the server's certificate is present in the server's entry in an LDAP directory. This optional step provides one way of ensuring that the server's certificate has not been revoked. (For this to happen in the ad-hoc mode, the entry needs to be in local cache, or is exchanged via IR port. -Need better solution!)
9. The client first uses its private key and then uses the server's public key (from the server's certificate) to decrypt the double encrypted session key.
10. Both the client and the server now have the same session key, which is a symmetric key used to encrypt and decrypt information exchanged during the application session and to verify its integrity -- that is, to detect any changes in the data between the time it was sent and the time it is received over the connection.
11. The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished.
12. The server, upon the receipt of the message sent by the client in the above, sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished and the client has been successfully added to the group.
13. The server then multicasts to the group the information of the new member.
14. The handshake is now complete, and the client becomes a member of the secure multicast group and can begin secure communication with the group.

It's important to note that both client and server authentications involve encrypting some pieces of data with one key of a public-private key pair and decrypting it with the other key:

In the case of client authentication, the client encrypts some random data with the client's private key -- that is, it creates a digital signature. The public key in the client's certificate can correctly validate the digital signature only if the corresponding private key was used. Otherwise, the server cannot validate the digital signature and the session is terminated. Also the server encrypts the session key with the client's public key, only the corresponding private key can correctly decrypt the secret, so the server has another assurance that the identity associated with the public key is in fact the client with which the server is connected. Otherwise, the client cannot decrypt the session key, and will send back the notification of completed handshake encrypted with the session key.

In the case of server authentication, the server encrypts a piece of data with the server's private key, that is, it creates a digital signature. The public key in the server's certificate can correctly validate the digital signature only if the corresponding private key was used. Otherwise, the client cannot validate the digital signature and the session is terminated. In addition, the server also encrypts the session key with its private key, only the corresponding public in the certificate can correctly decrypt the session key, so the client has some assurance that the session key is indeed sent by the server.



## **A.2 New member authenticate to group (client authentication)**

As explained in step 1 of the handshake section, the client sends the server a certificate to authenticate itself. The server uses the certificate in step 2 to authenticate the identity the certificate claims to represent.

When the user wants to join a group, he/she selects the targeted group's published group name or address from a list in his/her UI, sends the join request by pressing a button or similar action. For the underlying client program, along with the join request, it also sends over enough additional information for the server to authenticate itself.

To authenticate the binding between a public key and the client identity by the certificate that contains the public key, the server (group leader) must receive a "yes" to each of the following five questions. Step 6 is optional.

1. **Is today's date within the validity period?** The server checks the client certificate's validity period. If the current date and time are outside of that range, the authentication process won't go any further. If the current date and time are within the certificate's validity period, the client goes on to step 2.
2. **Is the issuing CA a trusted CA?** Each computer maintains a list of trusted CA certificates. This list determines which client certificates the server will accept. If the distinguished name (DN) of the issuing CA matches the DN of a CA on the list of trusted CAs, the answer to this question is yes, and the client goes on to step 3. If the issuing CA is not on the list, the client will not be authenticated unless the server can verify a certificate chain ending in a CA that is on the list.
3. **Does the issuing CA's public key validate the issuer's digital signature?** The server uses the public key from the CA's certificate (which it found in its list of trusted CAs in step 2) to validate the CA's digital signature on the client certificate being presented. If the information in the client certificate has changed since it was signed by the CA or if the CA certificate's public key doesn't correspond to the private key used by the CA to sign the client certificate, the server won't authenticate the client's identity. If the CA's digital signature can be validated, the server treats the client's certificate as a valid "letter of introduction" from that CA and proceeds. At this point, the server has determined that the client certificate is valid. Go to step 4.
4. **Does the DN (distinguished name) in the client's certificate match the user name he/she claims?** This step confirms that the client is the one who claims. (This step is hard to concisely define and need more work as we can't associate IP address to a personnel). If it matches, proceed; otherwise terminate.
5. **Does the client's public key validate the client's digital signature?** The server checks that the client's digital signature can be validated with the public key in the certificate. If so, the server has established that the public key asserted to belong to the client matches the private key used to create the signature and that the data has not been tampered with since it was signed.
6. **Is the authenticated client authorized to join the group?** The server checks the clients DN against a list of names that are authorized to join the group. This is an optional step that is suitable for some business meeting application scenario.

**The client (new member) is authenticated.** The server proceeds with the handshake. If the server doesn't get here for any reason, the client identified by the certificate cannot be authenticated, and the leader will be warned of the problem

After the steps described here, the server must successfully receive the notification sent from the client encrypted with the session key that says the client also has complete authentication and then really add the new member to the group; otherwise the session will be terminated. This provides additional assurance that the identity associated with the public key in the client's certificate is in fact the client who requested to join the group since the session key is sent to the client encrypted with the client's public key.

### **A.3 Group Leader authenticates to new member and session key distribute**

This should be similar to that of section 3. For the most parts just switch the role of server and client. I will wait until a late time to define the differences.

## **Appendix B**

### **Manually Validating Schannel Credentials in Win32 API**

By default, Schannel validates the server certificate by calling the WinVerifyTrust function; however, if you have disabled this feature using the ISC\_REQ\_MANUAL\_CRED\_VALIDATION flag, you must validate the certificate provided by the server that is attempting to establish its identity.

To manually validate the server certificate, you must first get it. Use the QueryContextAttributes function and specify the SECPKG\_ATTR\_REMOTE\_CERT\_CONTEXT attribute value. This attribute returns a CERT\_CONTEXT structure containing the certificate supplied by the server. This certificate is called the leaf certificate because it is the last certificate in the certificate chain and is farthest away from the root certificate.

Using the leaf certificate you must verify the following:

The certificate chain is complete and the root is a certificate from a trusted certification authority (CA).

The current time is not beyond the begin and end dates for each of the certificates in the certificate chain.

None of the certificates in the certificate chain have been revoked.

The depth of the leaf certificate is not deeper than the maximum allowable depth specified in the certificate extension. This check is only necessary if there is a depth specified.

The usage of the certificate is correct, for example, a client certificate should not be used to authenticate a server.

For server authentication, the server identity contained in the server's leaf certificate matches the server that the client is attempting to contact. Typically, the client will match some item in the certificate's Subject Name field to the server's IP address or DNS name.