8

JXTA and Security

IN THIS CHAPTER

- Importance of Security
- Security in Multifaceted
- Security in P2P Networks
- JXTA Platform Security
- JXTA Security Requirements
- The Cryptographic Toolkit
- Security Issues and Solutions
- Trust in P2P Systems
- P2P Security Models
- Summary

By Navaneeth Krishnan

Security is unquestionably one of the major concerns in any networked environment, and peer-to-peer networks are no exception. Needless to say, the success of your application will largely depend on how well you address your security concerns. Designing a suitable security model may prove to be one of the greatest challenges to a P2P application designer.

In this chapter, we will take a look at the various security concerns that might arise when developing P2P applications. We will go into the details of how JXTA strives to provide basic security primitives to help you build secure P2P applications. Towards the end, we will look at security issues in two popular P2P networks—Freenet and Napster.

Importance of Security

The importance of security in P2P applications cannot be underestimated. In fact, security is one of the major impediments to the widespread adoption of P2P, especially in corporate environments. There are basically two reasons for this.

The first reason is purely psychological. P2P gained widespread acknowledgement due to applications like Napster, Gnutella, and so on, which were also notorious for promoting and enabling unlawful activities, such as copyright violations. This resulted in an initial “pirate-to-pirate” image for the technology, and that has been a bit difficult to get away from.

The second is purely technical. Unlike security models for centralized systems, those for decentralized ones are much harder to implement.

However, distributed security is expected to change with the growth of the technology. Security models for P2P networks are predicted to mature with time and are likely to catch up with their client/server counterparts sooner than later.

Security is Multifaceted

The word security is so broad that it can be subject to a lot of interpretations (and misinterpretations). Security needs may vary from application to application. What may be desirable in one may not be desirable in another. Consider an application that enables anonymous file sharing, such as Freenet, and compare it with an enterprise application that enables collaboration between various people in the organization. The security model for the two may be strikingly dissimilar, and desirable features could be just the opposite. While the first application may be concerned about the anonymity of its users, the second may be concerned about the authenticity of its users.

The point is that P2P as such does not dictate any security features. It is up to the applications to implement security frameworks to suit their requirement.

Security Attacks in P2P Networks

Most, if not all, of the security concerns in a traditional client/server environment hold good for the P2P environment as well. In addition, P2P applications may introduce a whole new bunch of security concerns. For example, an employee can knowingly or unknowingly share confidential information to a file-sharing network like Gnutella and thus make it publicly available to the whole world.

While the above-mentioned concern may have to do more with company policy or usage of the technology rather than the technology itself, we must understand that such concerns also exist. For our discussion, however, we will be concerned more about the technical aspects of security.

Security attacks in P2P systems (as a matter of fact, in all networked environments) can be classified into two broad categories—active and passive network attacks.

Active Network Attacks

Active attacks are those in which the attacker is an active participant. The active attacker is usually in an aggressive mode. Different types of active attacks are possible:

- *Masquerades*—These are attacks in which the attacker pretends to be someone he is not. Most often, the attacker pretends to be some valid or privileged entity.

- *Man-in-the-middle*—As the name suggests, in this type of attack, the attacker intercepts the communication between two network nodes. The attacker may intend to modify or corrupt the information flow.
- *Playback or replay Attacks*—Such an attack usually involves capturing an exchange of information between two nodes and repeating the exact steps again to make it look like another genuine conversation.

Passive Network Attacks

Passive network attacks are those in which the attackers are predominantly in an inert state. The most significant form of passive attack is eavesdropping:

- *Eavesdropping*—Generally involves the silent capture of data by the attacker.
- *Traffic analysis*—Here, the attacker not only captures the data but also tries to learn more by analyzing the data.

Usually, passive attacks are precursors to active attacks. For example, an attacker may first be passive in a network and sniff all the traffic between Peer A and Peer B. After Peer A has left, the attacker may communicate with Peer B by replaying the exact data that Peer A had initially transmitted. This is a case of an Eavesdropping as a precursor to a replay attack.

While it might not be practically possible to entirely eliminate the chances for security attacks, what needs to be done is to minimize possibility of such attacks.

JXTA Platform Security

The JXTA Java Binding by itself contains many built-in security features that can enhance applications built over it. Even though these are by no means sufficient for building a full-fledged secure application, they can potentially provide a base over which secure JXTA applications can be built.

Currently, the following security features are provided by the JXTA platform:

- TLS as a Secure Transport layer (TLS)—TLS (RFC 2246), also known as Secure Sockets Layer (SSL) V3.1, is based on public key technology. The JXTA platform provides TLS as a medium of secure communications. Applications can exploit TLS capabilities of the platform by using secure pipes, which internally use TLS, to guarantee safety against passive attacks.
- *Peer certificates*—The TLS layer requires the use of certificates to enable its functioning. Consequently, each peer generates its own certificate and acts as its own Certification Authority (CA). This certificate, called the root certificate, is used to sign service certificates that the peers issue for each service that it supports.

The root certificate is distributed along with the peer's advertisement. Therefore, every other peer can always verify that an advertisement is indeed from the peer who claims to have issued it.

- *Personal security environment*—Every peer is protected by a peer ID and password. This is used to decrypt the private key to a user's personal security environment. This acts as a first line of defense against a local attacker (an attacker having physical access to the machine running the JXTA Peer).

JXTA Security Requirements

The JXTA platform depends on the following packages and APIs for its security requirements:

- *PureTLS*—Pure TLS (`http://www.rtfm.com/puretls/`) is an open-source Java implementation of the TLS protocol. It is used to achieve end-to-end privacy between peer communications.
- *Cryptix 3*—Cryptix (`http://www.cryptix.org`) is a clean room implementation of Sun's JCE 1.1. JCE stands Java Cryptographic Extensions and provides standard interfaces for cryptographic algorithms and services. It is used by PureTLS for various algorithms.
- *Cryptix ASN.1 Kit*—ASN.1 stands for Abstract Syntax Notation One (`http://sourceforge.net/projects/cryptix-asn1`). Simply put, it is a language that allows definitions of various data types, such as integers, strings, sequences, and so on. Due to its simplistic encoding scheme, it can be used to message between different network applications. The ASN.1 kit can be used to use this notation in Java programs. It is used by the JXTA platform to support X509V3 certificates.
- *Bouncy Castle Crypto APIs*—Another open source JCE implementation used to generate certificates(`http://www.bouncycastle.org`).

The Cryptographic Toolkit

In addition to the few basic security features that come with the JXTA platform, the JXTA Security Project provides a toolkit with a basic set of security algorithms that can be used in JXTA applications.

Because JXTA is targeted for almost every device with a digital heartbeat, the security had to reflect this design philosophy. It had to be as simplistic and minimal as possible and also cater to resource-constrained environments. Therefore, the JXTA Security Toolkit was designed based on the JavaCard 2.1 security model.

JavaCard is the technology that powers Java on smart cards that may probably be the most resource constrained Java environment.

JXTA Security Suites

A *security suite* represents a full-fledged security system. It consists of various security algorithms and services that can be used by the clients of the suite. One may visualize a suite as a factory of security algorithms. Figure 8.1 shows the `JxtaCrypto` interface that is used to access each of the services provided in the security package. We will discuss each of these interfaces in the following paragraphs.

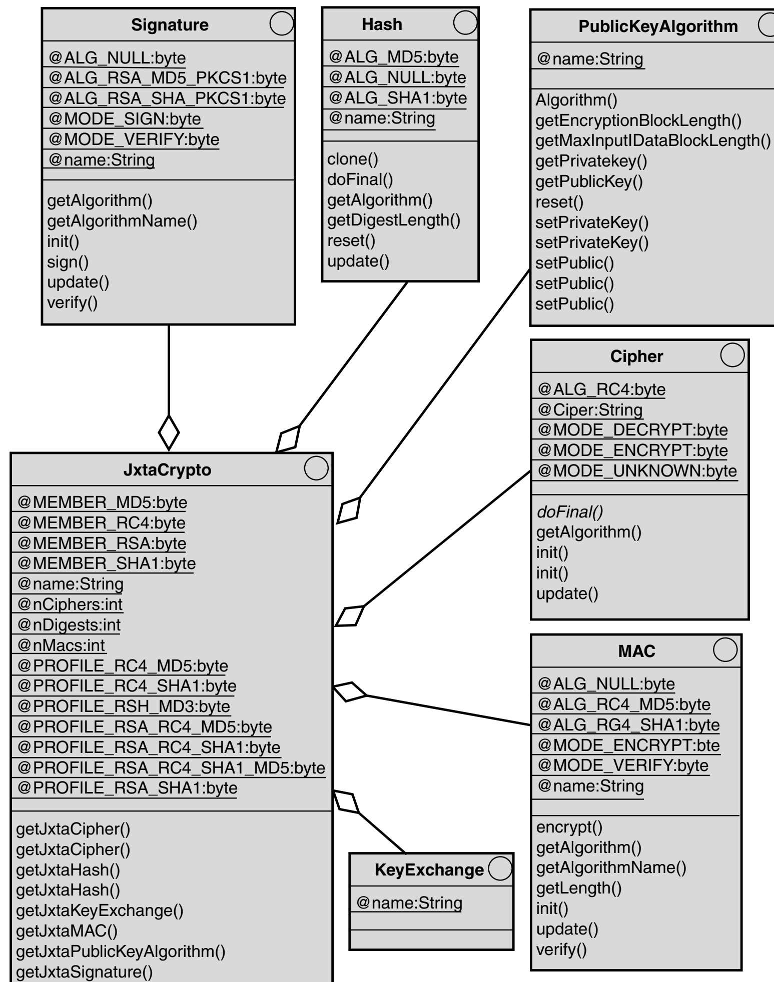

FIGURE 8.1 UML relationships of the `JxtaCrypto` interface and key interfaces in the JXTA Crypto package.

JXTA Security defines an interface that all suites have to adhere to, the `JxtaCrypto` interface is shown in the following listing. Each method returns a class that implements a standard interface. The interface is used to retrieve the supported versions of these interfaces.

```
public interface jxta.security.crypto.JxtaCrypto {
  public Cipher getJxtaCipher();
  public Cipher getJxtaCipher(byte type) throws CryptoException;
  public Hash getJxtaHash();
  public Hash getJxtaHash(byte type) throws CryptoException;
  public KeyExchange getJxtaKeyExchange();
  public MAC getJxtaMAC();
  public MAC getJxtaMAC(byte type) throws CryptoException;
  public PublicKeyAlgorithm getJxtaPublicKeyAlgorithm();
  public Signature getJxtaSignature();
}
```

The idea behind having a standard interface is that multiple implementation of the security system, probably from different providers, can be easily plugged into the system.

Nonetheless, JXTA Security also provides a default implementation of the `JxtaCrypto` called the JXTACrypto Suite (`net.security.impl.crypto.JxtaCryptoSuite`).

The JXTACrypto Suite

Being the default security suite of the JXTA platform, the JXTACrypto Suite supports various types of cryptographic operations, such as encryption, hashing, and so on. Now, each of these types of operations can be implemented using different algorithms. For example, hashing can be done using either an MD5 or an SHA algorithm. For this purpose, JXTA defines what is called an algorithm profile.

Algorithm Profiles

An *algorithm profile* can be defined as one particular combination of algorithms. This means that profiles will define one specific algorithm for encryption, one specific algorithm for hashing. For example, the `PROFILE_RSA_RC4_SHA` uses the RSA algorithm for asymmetric encryption, RC4 algorithm for symmetric encryption, and SHA for hashing.

For a detailed list of supported profiles, refer to Table 8.1.

TABLE 8.1 Basic JXTA Algorithm Profiles

Profile Name	Cipher	Public Key	Hash
PROFILE_RSA_SHA1	N.A.	RSA	SHA1
PROFILE_RSA_MD5	N.A.	RSA	MD5
PROFILE_RC4_SHA1	RC4	N.A.	SHA1
PROFILE_RC4_MD5	RC4	N.A.	MD5
PROFILE_RSA_RC4_SHA1	RC4	RSA	SHA1
PROFILE_RSA_RC4_MD5	RC4	RSA	MD5
PROFILE_RSA_RC4_SHA1_MD5	RC4	RSA	SHA1, MD5

Security Issues and Solutions

In this section, we will look into the details of various security issues that P2P applications can face and the relevant APIs that the Crypto Toolkit provides to address them.

To better illustrate the usage of JXTA Crypto APIs, you will build a `SecureMembershipService`. You will also reuse the `GroupManager` that you built in Chapter 6, "Working with Groups," by plugging in this membership service.

Before we proceed, it is important to note that both the peer and the membership authenticator reside on the same physical machine in this example. Therefore, secure exchanges between them may not make much sense in the real world. The intention is to make it simpler to understand the APIs. The techniques that we discuss can be easily extended to apply to remotely communicating entities.

Privacy

Privacy means that data that is transmitted from one peer to the other must not be comprehensible to any other peer in the network. The most direct way to address this issue is by encryption.

Encryption

Encryption is the process of converting data (plain text) into a nonsense form (cipher text) so that it becomes intelligible only to authorized parties. Encryption is based on *keys*. Just like keys opening real world locks, using encryption keys one can open encrypted (locked) text.

Based on the use of keys, encryption can be classified into two broad categories—symmetric encryption and asymmetric encryption.

Symmetric Encryption

Symmetric encryption is also called *single key encryption*. It is the most popularly used encryption type today and was the only one available until asymmetric encryption was introduced.

In a symmetric cryptographic algorithm, both the encryption and decryption processes use the same key. Consequently, both the sender and receiver of the message must have a shared secret key by which they encrypt and decrypt messages sent between each other. In Figure 8.2, you can see a message being passed between our users Alice and Bob. They are using a single key to encrypt and decrypt a message. Note that the key was exchanged between them via another channel. Note also that Alice handed the key directly to Bob via some method (ideally in a way that prevents others from seeing the key, because anyone can decrypt the message if they had it).

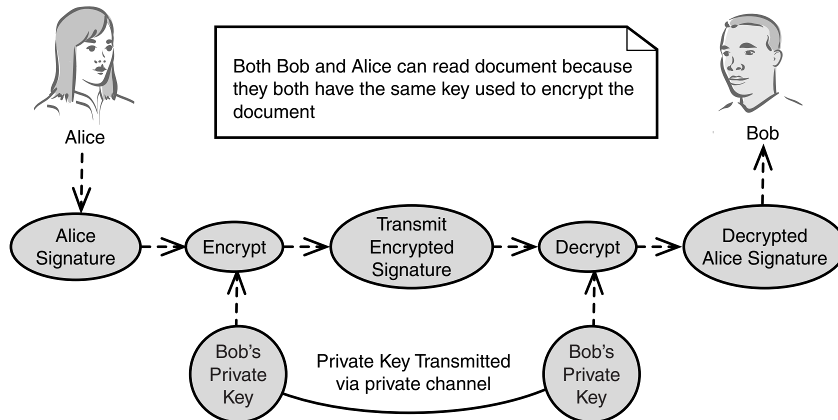

FIGURE 8.2 Symmetric encryption.

While this method is a simple but powerful way of secure communication, it has its disadvantages because of that single key. It requires a secret key be used for any pair of communicating parties. Although this is practical for small-scale communications, it is not scalable for a large-scale system where a large number of users intercommunicate if all of the pairs of users have unique keys so that no others, even in the same group, could pass secret messages. However, it is a good method if a group of users needs to send messages that can be read by anyone with a key.

The other concern is about sharing the key itself. How do a sender and a receiver share an encryption key securely in the first place? It would defeat the purpose if the insecure communication channel itself were used. Therefore, such systems depend on an out-of-band secure channel to share the key. And if such a secure channel was present, why not use that channel for data transfer as well?

However, symmetric encryption is still widely used. Compared to other forms of encryption, symmetric encryption is very fast, so it becomes increasingly important as the size of the data to be encrypted increases. Consequently, symmetric encryption is chosen when data to be encrypted is large.

Two of the most popular symmetric encryption algorithms are RC4 and DES. DES stands for *Data Encryption Standard* and was published by FIPS 46 (Federal Information Processing Standards Publication). RC4 is owned by RSA.

Symmetric Encryption Using the JXTA Crypto APIs

The JXTA Security Project defines a parent interface for all symmetric cryptographic algorithms. The `init` method initializes the instance of `Cipher` with the key and parameters. The `update` method performs the encryption and decryption when all input data is not available. When the message is complete, the `doFinal` method is called. The `doFinal` method performs the encryption or decryption. The `getAlgorithm` method simply returns a code that matches the constant used to select the algorithm:

```
public interface net.jxta.security.cipher.Cipher extends Description{
  public void init( Key theKey
                          , byte theMode) throws CryptoException;

  public void init( Key theKey
                  , byte theMode
                  , byte[] bArray
                          , int bOff,int bLen) throws CryptoException;

  public byte getAlgorithm();

  public int update( byte[] inBuff
                   , int inOffset
                   , int inLength
                   , byte[] outBuff
                           , int outOffset )throws CryptoException;

  public abstract int doFinal( byte[] inBuff
                             , int inOffset
                             , int inLength
                             , byte[] outBuff
                                     , int outOffset) throws CryptoException;
}
```

The `Cipher` interface extends the `Description` interface. The `Description` interface is the parent interface for all algorithms provided by the JXTA Security framework and provides an interface to get the name to control debugging.

```
public interface jxta.security.util.Description{
        String getAlgorithmName();
        public void setDebug();
        public void clearDebug();
}
```

Currently, only the RC4 algorithm is supported. This is made available by the `jxta.security.impl.cipher.RC4Cipher` class.

Encrypting Documents Using the `EncryptedDocumentsFactory`

You may recall that the first step towards joining a peer group is by invoking an `apply` method on the Membership Service. This is done using an `AuthenticationCredential` that contains an `identityInfo` document.

The role of the `EncryptedDocumentsFactory` class we have written is to provide you with this document. Not only does this class act as a factory for a `StructuredDocument`, it also ensures that the document is encrypted using RC4 encryption.

The class itself is simple, with just one static method. Listing 8.1 shows the `EncryptedDocumentsFactory` class, and Figure 8.3 shows the UML diagram.

LISTING 8.1 `EncryptedDocumentsFactory.java getEncryptedDocument()` Method

```
public static StructuredDocument getEncryptedDocument() throws
                                jxta.security.exceptions.CryptoException{

  // Secret shared with the Membership Service
  String sharedSecret = "Shared Secret Key";
  // This is the data which we will encrypt
  String dataToBeEncrypted = "Secret Data";
  // First build a key using the KeyBuilder
  SecretKey secretKey=( SecretKey)KeyBuilder.buildKey(KeyBuilder.TYPE_RC4
                    , KeyBuilder.LENGTH_RC4
                    , false);

  // All Keys need to have a minimum
  // length for a successful encryption

  int minimumKeyLength = secretKey.getLength();
```

LISTING 8.1 Continued

```
// If our key does not fulfill the minimum
// length requirement,
// we need to pad zero byte values with it
// and make it long enough
byte[] keyArray;
if(sharedSecret.length()<minimumKeyLength){
  // In this case we have to pad the
  // key such that it has the minimum key length.
  // The new byte[] ensures that all bytes are
  // initialized to 0X00
  keyArray = new byte[minimumKeyLength];
              System.arraycopy( sharedSecret.getBytes()
                , 0
                , keyArray
                        , 0
                , sharedSecret.length());
} else {
  // The key is more that the required length.
  // Hence no concerns
  keyArray =sharedSecret.getBytes();
}
// Set the secret key with our value
secretKey.setKey(keyArray, 0);
// From the JXTACrypto Suite , get the RC4 cipher algorithm
JxtaCrypto crypto = new JxtaCryptoSuite(JxtaCrypto.MEMBER_RC4
                                       , null
                                       , (byte)0
                                       , (byte)0);
Cipher rc4Algorithm = crypto.getJxtaCipher();

//Initialize the Algorithm with the key and required mode
rc4Algorithm.init(secretKey, Cipher.MODE_ENCRYPT);
// Create a new byte[] to store the output
byte[] outputBuffer = new byte[dataToBeEncrypted.length()];
// The actual encryption is done by this method
rc4Algorithm.doFinal( dataToBeEncrypted.getBytes()
                    , 0
                    , dataToBeEncrypted.length()
                    , outputBuffer
                    , 0);
String encryptedData = new String(outputBuffer);
```

LISTING 8.1 Continued

```
  LOG.debug("ENCRYPTED DATA => "+encryptedData);
  // Now we have successfully encrypted the method name.
  // Return a Structured Document using this method name
  MimeMediaType type = new MimeMediaType("text","xml");
  StructuredDocument doc
          = StructuredDocumentFactory.newStructuredDocument(type,"Apply");
  Element e = doc.createElement( "Data",encryptedData);
  doc.appendChild( e );
  return doc;
}
```

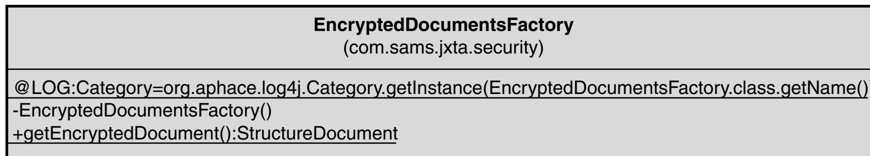

FIGURE 8.3 The UML for the `EncryptedDocumentsFactory` class.

Note how the `jxta.security.impl.cipher.KeyBuilder` class is used to generate an RC4 secret key. This key is just a placeholder containing no data. You need to initialize the key with some data.

The key generation routine used in Listing 8.1 is a bit eccentric. You have used the `sharedSecret` string to generate the data required to initialize the key:

```
secretKey.setKey(keyArray, 0);
```

However, this is never the case in real-world cryptographic systems. Symmetric keys are always generated randomly, and the strength of the encryption will depend on how strong the random number is. It should not be possible for an attacker to discover the key by exactly duplicating the process of random number generation.

Consequently, what you need is a generator to generate secure random numbers that can be used for encryption. The `JRandom` class fulfills this need:

```
public class jxta.security.impl.random.JRandom extends java.util.Random
```

It can be used to generate pseudo-random (theoretically, these are not “random”) data. The initial seed—the input random data that the generator takes—should be set to better values to get more secure numbers. The `JRandom` class can be used as shown:

```
try{
  JRandom jrandom = new JRandom();
} catch(CryptoException crypto){
   // Handle error
}
byte[] randomData = new byte[10];
// Fill the array with random data
jrandom.nextBytes(randomData);
```

Decrypting RC4 Encrypted Documents

The RC4 encrypted document is decrypted by the membership service `SecureMembershipService` shown in Listing 8.2 and Figure 8.3.

LISTING 8.2 `SecureMembershipService.java` `apply()` Method

```
public Authenticator apply( AuthenticationCredential unsubscribedCredential )
    throws PeerGroupException, ProtocolNotSupportedException{

  String sharedSecret = "Shared Secret Key";
  String method = unsubscribedCredential.getMethod();
  if( (null != method) && !"Apply".equals( method ) )
      throw new ProtocolNotSupportedException(
  "Method not recognized : Required \"Apply\" ");
  // First Extract the "Data" from the document
  StructuredDocument doc =
      (StructuredDocument)
        ➥unsubscribedCredential.getIdentityInfo();
  Enumeration enum = doc.getChildren();
  Element element = (Element) enum.nextElement();
  String encrypteddata =(String) element.getValue();

  // Now to decrypt this data, we must use the shared secret
  try{
    // First build a key using the KeyBuilder
    // with RC4 as the algorithm
    SecretKey secretKey
         =(SecretKey)KeyBuilder.buildKey ( KeyBuilder.TYPE_RC4
                                         , KeyBuilder.LENGTH_RC4
                                         , false);
    // All Keys need to have a minimum length
    // for a successful encryption
    int minimumKeyLength = secretKey.getLength();
```

LISTING 8.2 Continued

```
// If our key does not fulfill the minimum
// length requirement, we need to pad zero byte values
//  with it and make it long enough
byte[] keyArray;
if(sharedSecret.length()<minimumKeyLength){
  // In this case we have to pad the key such that it has the
  // minimum key length. The new byte[] ensures that all bytes are
  // initialized to 0X00
  keyArray = new byte[minimumKeyLength];
  System.arraycopy( sharedSecret.getBytes()
                  , 0
                  , keyArray
                  , 0
                  , sharedSecret.length());
} else {
  // The key is more that the required length.
  // Hence no concerns
  keyArray =sharedSecret.getBytes();
}
// Set the secret key with our value
secretKey.setKey(keyArray, 0);
// From the JXTACrypto Suite , get the RC4 cipher algorithm
JxtaCrypto crypto = new JxtaCryptoSuite( JxtaCrypto.MEMBER_RC4
                                       , null
                                       , (byte)0
                                       , (byte)0);
Cipher rc4Algorithm = crypto.getJxtaCipher();

//Initialize the Algorithm with the key and required mode
rc4Algorithm.init(secretKey, Cipher.MODE_DECRYPT);
byte[] encrypteddataArray =encrypteddata.getBytes();
        byte[] decryptedData = new byte[encrypteddataArray.length];
rc4Algorithm.doFinal( encrypteddataArray
                    , 0
                    , encrypteddataArray.length
                    , decryptedData
                    , 0);
String decryptedString = new String(decryptedData);
LOG.debug("DECRYPTED STRING IS =>"+ decryptedString);
```

LISTING 8.2 Continued

```
    if(!decryptedString.equals("Secret Data")){
                      throw new PeerGroupException ("Data not properly encrypted
      !");
    }
  } catch(jxta.security.exceptions.CryptoException cryptoException){
    LOG.error("Error in Apply Process",cryptoException);
        throw new PeerGroupException("Failure in Apply");
  }
return new SecureAuthenticator(this,unsubscribedCredential);
}
```

As you may have noticed, the same `doFinal` method of the `Cipher` class was used for both encryption and decryption. It all depends on how the algorithm is initialized. For encryptions, initialization is done in the `MODE_ENCRYPT`. For decryptions, a `MODE_DECRYPT` is used.

Asymmetric Encryption

Asymmetric encryption or *public key encryption* is increasingly becoming popular as a secure means of many-to-many communication. It relies on two different keys (therefore, asymmetric) for secure interactions. Every participating user has a pair of keys generated by an algorithm. The keys have a unique mathematical property that any message encrypted by one can only be decrypted by the other.

The user is free to designate one of these as a public key and the other as a private key. The public key is used to represent the user and can be widely distributed on whatever channels the user wants. The user keeps the private key confidential. Any sender who wants to communicate with the user can use the users' public key to encrypt the message. Rest assured, only the user will be able to decrypt the message. Similarly, if the user wants to respond to another user, he or she can encrypt the other party's public key for encrypting the message. This can go a step further, and the user can encrypt the message with his or her private key and the receiver's public key to ensure that not only does the proper person receive the message but also for the receiver to recognize the source of the message.

An example of how this type of encryption works is shown in Figure 8.4. First, our user Alice encrypts the document meant for Bob with Bob's public key. The message is transmitted and Bob decrypts the message with his private key. The process is identical for users, other than Alice, that have access to Bob's public key. The decryption can only be done by Bob—as long as Bob does not let anyone copy his private key.

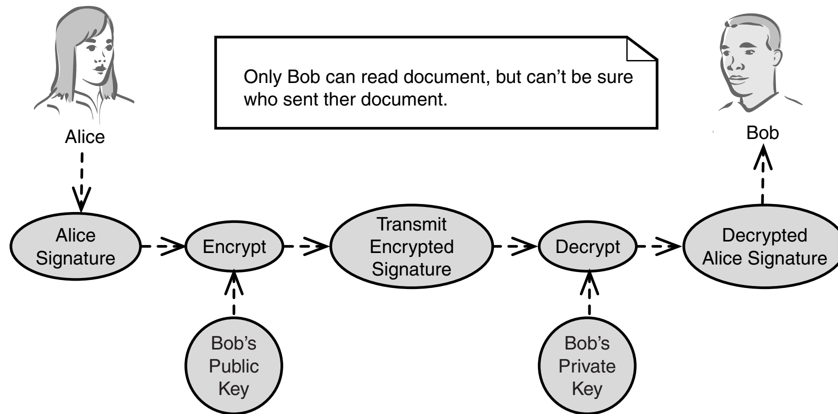

FIGURE 8.4 Asymmetric encryption (public key encryption).

Because the encryption is reversible (that is, the private key can be used to encrypt and the public key to decrypt, messages can be created that essentially prove they came from the owner of the key). In other words, as shown in Figure 8.5, if the message is encrypted by Alice using her private key, Bob can decrypt it using Alice's public key. If, as shown in this figure, the document is encrypted first with Bob's public key and then Alice's private key, the document is only decryptable by Bob and, because only Alice's public key can encrypt these contents, only Alice could have sent the message.

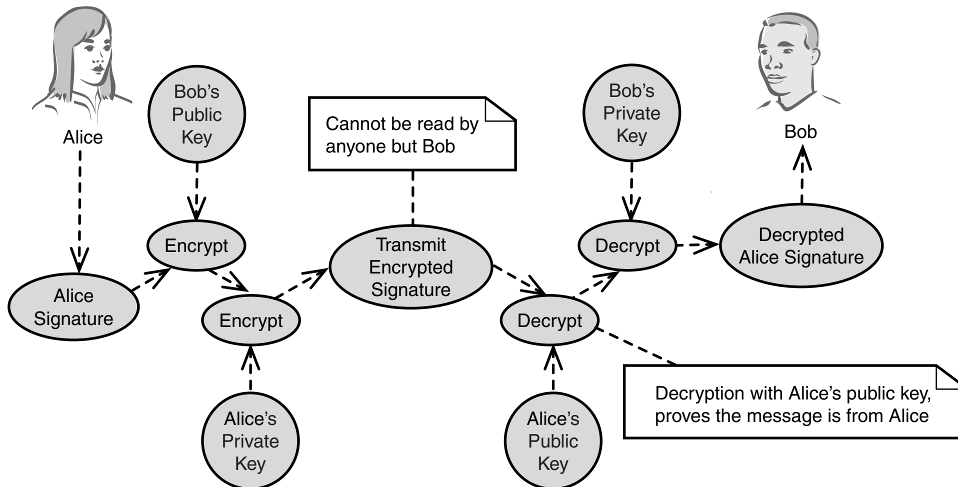

FIGURE 8.5 Asymmetric authentication.

The approach solves the main problem with symmetric key encryption caused by the exchange of what is here a private key because the private keys are never exchanged. The question of how to securely transmit private keys does not arise. The encryption/decryption processes does have a downside because it consumes more processing time and memory for calculations than symmetric key encryption. Unless you are transferring large amounts of data, the time to run the algorithms is livable for most applications (seconds to a few minutes depending on your hardware and the size of the message).

One good approach is to use a combination of symmetric and asymmetric keys. Public key encryption can be used to securely exchange a set of symmetric keys. After this is done, symmetric encryption can be used to encrypt all subsequent exchanges, so you can benefit from the best of both worlds. SSL and TLS, for example, use such an approach.

Asymmetric Encryption Using the JXTA Crypto APIs

All public key algorithms implement the `PublicKeyAlgorithm` interface shown in the following code. The methods are categorized into the setters and getters for the keys, a rest to reset the keys and the `algorithm` (this is not a typo—the method name is capitalized) method that is used to decrypt or encrypt a message.

```
public interface jxta.security.publickey.PublicKeyAlgorithm
                                                 extends Description {
  public void reset();
  public int getMaxInputDataBlockLength();
  public int getEncryptionBlockLength();
  public void setPublicKey() throws CryptoException;
  public void setPublicKey(Object publickeyData)
  public void setPublicKey(byte[] nModulus)
  public void setPrivateKey() throws CryptoException;
  public void setPrivateKey(Object privatekeyData) throws CryptoException;
  public Object getPublickey() throws CryptoException;
  public Object getPrivatekey() throws CryptoException;
  public byte[] Algorithm(byte[] data,int offset,int length,
                              byte type,boolean encrypt) throws CryptoException;

}
```

The JXTA platform supports RSA with PKCS#1 padding. *Padding*, which is the process of adding random data to the data to be encrypted, is usually used to reduce the effectiveness of attacks against the encrypted dat. By adding padding, all the messages have the same length and the attackers cannot find any information about the data because “Hello” has the same length as “Hi everybody.”

Hashing

A hash function is used to compact data in such a way that it is verifiable but non-reversible. In other words, a given piece of information must always hash to the same value but, using this value, it must be very difficult to reconstruct the original information. Another criterion for a hash function is that it must be highly improbable (practically impossible) to find two pieces of information that produce the same hash result. In addition, a good hash algorithm gives very different hash messages for any two pieces of data that are very similar.

Hashing can be considered as an encryption routine without a corresponding decryption routine. At first glance, you may tend to doubt the usefulness of such a function. Why would you want to encrypt, if there is no way to decrypt?

The answer is straightforward. Hashes can act as a powerful way of verification. Hashes can act as data fingerprints, being practically unique for all kinds of data.

Even though hashes cannot be used as a direct encryption routine, they are extremely useful for authentication. One of the simplest examples of using a hash function can be found in most operating systems. Almost all operating systems, including flavors of Unix and Microsoft Windows, use hashes for user authentication. Passwords are not stored directly; only their hashes are stored. Whenever a user tries to log in, his or her password is taken and hashed. This is then compared with the stored result to authenticate him or her. The advantage is that even if one has access to the password file, it is practically impossible to retrieve the user passwords (unless the ID and password are found in other ways via keyboard snooping or other means—no solution is perfect).

Hashing Using the JXTA Crypto APIs

All hashing algorithms in JXTA are implementations of the `Hash` interface:

```
public interface jxta.security.hash.Hash extends Description
```

Currently, JXTA supports two different hashing algorithms—MD5 and SHA-1 (shown in the following code fragment). SHA-1 produces a 160-bit (20 byte) message digest. Although slower than MD5, this larger digest size makes it stronger against brute force attacks. MD5 is a 128-bit (16 byte) message digest makes it a faster implementation than SHA-1.

```
public class MD5Hash implements jxta.security.hash.Hash ;
public class SHA1Hash implements jxta.security.hash.Hash ;
```

Using the MD5Hash Algorithm

The `SecureAuthenticator` class in Figure 8.6 is the authenticator used by the `SecureMembershipService` (Figure 8.7) in the `isReadyForJoin` method shown in Listing 8.3. This class demonstrates the use of the `MD5Hash` algorithm. The user is prompted to fill up the authenticator by providing a password. The class is designed to accept only one password that, incidentally, has the value `password`. The user's password is then hashed and compared to the MD5 of the valid password. The user is allowed to join only if the password is correct.

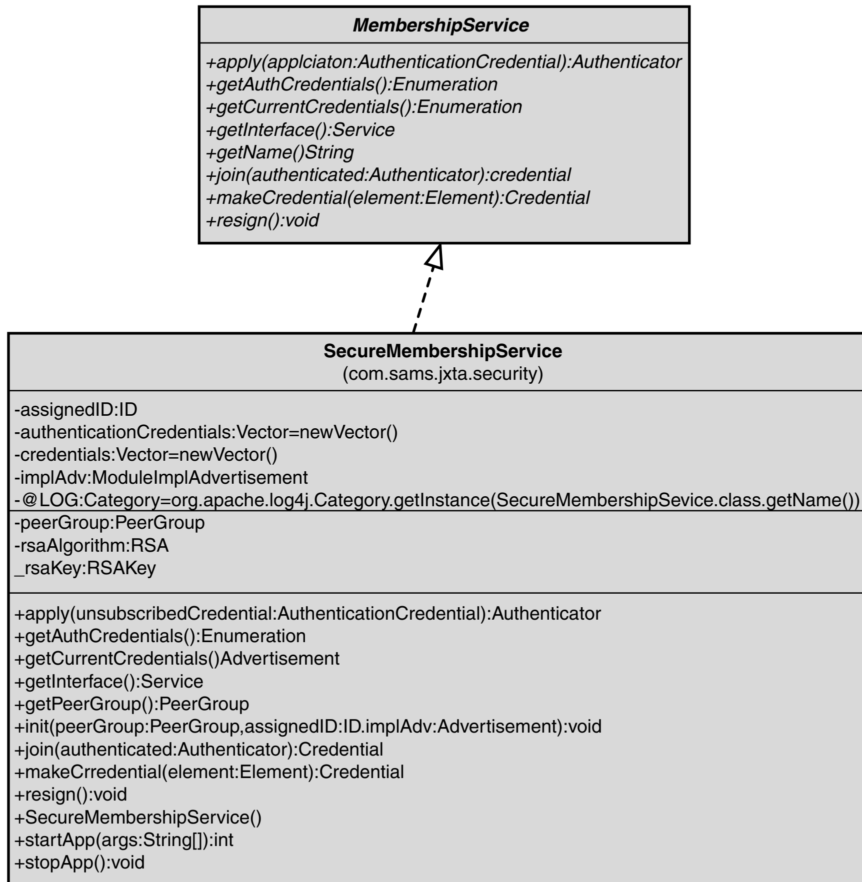

FIGURE 8.6 UML for the *`SecureMembershipService`* class.

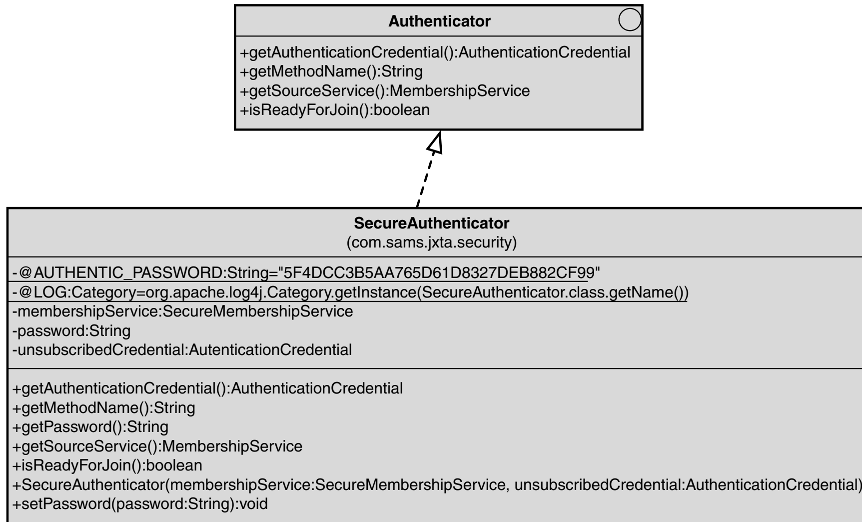

FIGURE 8.7 UML for the *SecureAuthenticator* class.

LISTING 8.3 SecureAuthenticator.java isReadyToJoin() Method

```
public boolean  isReadyForJoin(){
  // we calculate the MD5 hash of the password
  // entered by the user
  // and compare it with the AUTHNETIC_PASSWORD.
  try{
    // Now to create the MD5 hash ,first get
    // a suite with the MD5 Algorithm
    JxtaCrypto crypto = new JxtaCryptoSuite( JxtaCrypto.MEMBER_MD5
                                        , null, (byte)0, (byte)0);
    // Now get the Hash Object
    Hash hash = crypto.getJxtaHash();
    // Get the digest length of the Hash
    int digestLength = hash.getDigestLength();
    byte[] passwordInBytes = password.getBytes();
    // Calculate the length needed by the output byte array
    int outputLength = (password.length() < digestLength
                        ? digestLength : password.length());
    // Create a new array to hold the output
    byte[] outputBytes = new byte[digestLength];
    // This is where the actual hashing is done
    hash.doFinal(passwordInBytes, 0, passwordInBytes.length, outputBytes, 0);
    // Base64 encode the result
    String finalPass
        = new String(jxta.security.util.Util.hexEncode(outputBytes));
```

LISTING 8.3 Continued

```
    // Compare it with the AUTHENTIC_PASSWORD
    // If they match , the authenticator is ready to join
    if(finalPass.equals(AUTHENTIC_PASSWORD))
    return true;
    else
    return false;

  } catch(CryptoException cryptoException){
    LOG.error("Exception in creating MD5 Hash",cryptoException);
  }
  return true;
}
```

Authentication

Authentication is an important requirement in almost all secure systems. It must be possible to

- Verify that the sender of the data is indeed who he claims himself to be.
- Ascertain that the data sent has not been modified in transit by a malevolent entity.
- Disclaim or repudiate messages that have not been received or sent by a particular entity.

Encryption may by itself provide authentication in some cases. Consider a readable text message encrypted by using the private key of the sender. The recipient can decrypt the message using the sender's public key. The recipient can also be ensured that it is, indeed, the same entity that sent the message, because no one else possess the private key to encrypt the text message. If an invalid key had been used to generate the encrypted data, it would not convert to the readable message after decryption.

This view is a bit short sighted because we have only considered an example of a readable message. What if the message itself is an arbitrary block of meaningless data? Trying to decrypt such an encrypted message with any key will lead to another block of data. How do you know that this is indeed legitimate?

From these discussions, you can safely assume that encryption will, indeed, provide authentication if the data transmitted and received is human-readable.

There are two major approaches to ensure authentication in secure systems—Message Authentication Code (MAC) and Digital Signatures.

Message Authentication Codes

A Message Authentication Code, popularly referred to as MAC, is a simple way to ensure authentication. MAC relies on symmetric key encryption; in other words, it assumes that both the sender and the receiver share a common secret key.

A simple way of generating a MAC is:

1. Concatenate the message with the secret key (or combine them in some other way.
2. Hash the resultant data to get the MAC.
3. Attach the MAC to the message to be sent.

The receiver can, in turn, receive the message and the MAC, recalculate the MAC from the message and secret key, compare it with the received MAC, and verify its authenticity.

MAC Algorithms in the JXTA Crypto Suite

All MAC algorithms are implementations of the `MAC` interface, shown in the following code. The `init` method sets the keys and type of encryption, while the `encrypt` method is used to create signatures. The `verify` method is used to verify a MAC for the `inBuff` against the signature in `sigBuff`. The `update` method is used to update the `inBuff`. Of course, `getAlgorithmName` and `getAlgorithm` are used to identify what the MAC implementation is supporting.

```
public interface jxta.security.mac.MAC extends Description {
  public String getAlgorithmName();
  public byte getAlgorithm();
  public int getLength();
  public void init( byte theMode
                , Key theKey
                , byte[] privateKey)throws CryptoException;
  public void update( byte[] inbuf
                  , int offset
                  , int length) throws CryptoException;
  public int encrypt( byte[] inbuff
                  , int offset
                  , int inLength,byte[] macBuff
                  , int macOffset)throws CryptoException;
  public boolean verify( byte[] inBuff
                     , int inOffset
                     , int inLength
                     , byte[] macBuff
                     , int macOffset
                     , int macLength) throws CryptoException;
}
```

JXTA supports two different types of MACs profiles in addition to the basic profiles:

- MACs using RC4 encryption and SHA1 hashing, denoted by an algorithm profile `ALG_RC4_SHA1`.

- MACs using RC4 encryption and MD5 hashing, denoted by the algorithm profile ALG_RC4_MD5.

You can get an instance of the MAC algorithm by using

```
JxtaCrypto crypto = new JxtaCryptoSuite( JxtaCrypto.PROFILE_RC4_SHA1
                                       , null
                                       , (byte)0
                                       , MAC.ALG_RC4_SHA1);
MAC mac = crypto.getJxtaMAC();
```

Digital Signatures
Digital signatures are another form of authentication. Many popular systems available today, such as Pretty Good Privacy (PGP), are based on digital signatures. Digital signatures, in turn, rely on the concept of asymmetric encryption. Figure 8.8 shows how Alice can create an encrypted signature that Bob can decrypt with Alice's public key to prove the signature was from Alice.

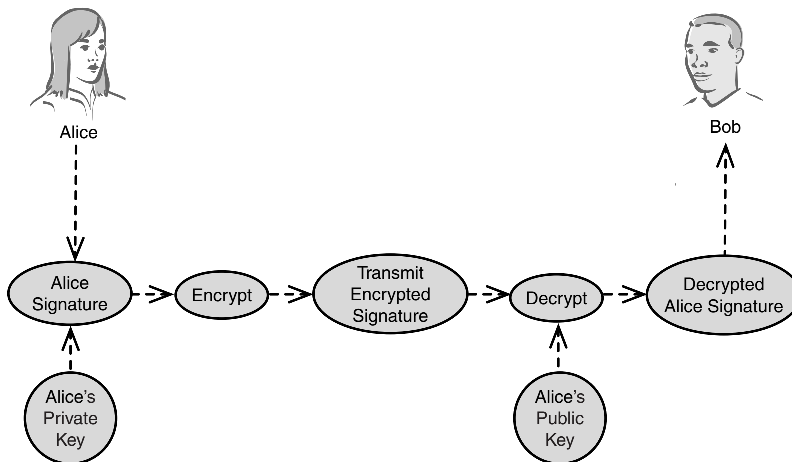

FIGURE 8.8 The example of the digital signature process.

Digital signatures can be used to sign documents by including the document in the signature or encrypting an already encrypted document (as in Figure 8.5). Alternatively, and most widely used, a checksum of a document is calculated and the checksum stored in the signature. after the receiver decrypts the signature, the checksum found in the message is compared to the checksum of the transmitted document. This is simple and fast because only a small checksum is processed with the crypto algorithms. Besides acting as a signature, this sum also proves that the document has not been modified since the document was signed.

Data can be digitally signed using the following simple steps:

1. Calculate the hash of the message to be signed.
2. Encrypt this hash with the private key.
3. The encrypted data obtained is the digital signature and can be attached with the document.

Signing Using the JXTA Crypto APIs

JXTA offers two algorithm profiles for digital signatures. `AlG_RSA_SHA1_PKCS1` uses the RSA algorithm with PKCS#1 padding as the encryption algorithm and SHA1 as the hash algorithm. `ALG_RSA_MD5_PKCS1` also uses RSA algorithm with PKCS#1 as the encryption algorithm but uses MD5 for the hashing.

All signature algorithms implement the `Signature` interface shown in the following vode. The methods are used to access information about the algorithm and to initialize it. The `sign` and `verify` methods are used to sign and verify messages:

```
public interface Signature extends Description{
  public String getAlgorithmName();
  public byte getAlgorithm();
  public void init(byte theMode) throws CryptoException;
  public void update(byte[] inbuf,int offset,
        ➥ int length)  throws CryptoException;
  public byte[] sign(byte[] inbuff, int offset,
        ➥ int inLength) throws CryptoException;
  public boolean verify(byte[] inBuff,int inOffset,
        ➥     int inLength,byte[] sigBuff,int sigOffset,
        [ccc]int sigLength) throws CryptoException;
}
```

The Secure Credential

On a successful apply to your `SecureMembershipService`, a `SecureCredential` is returned to the peer. We refer to the credential as secure because it contains a digital signature that can be verified by the peer. Figure 8.9 shows the UML diagram of the `SecureCredential` class. Listing 8.4 illustrates how the credential is signed in the `createSignedMessage` method.

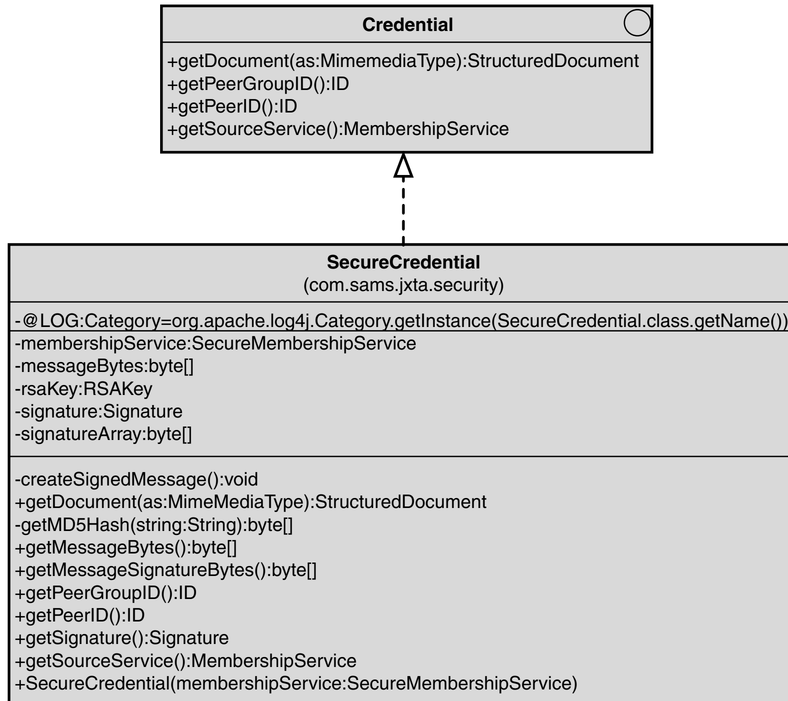

FIGURE 8.9 The UML for the *`SecureCredential`* class.

LISTING 8.4 `SecureCredential.java` `createSignedMessage()` Method

```
private void createSignedMessage(){
  String date = DateFormat.getDateTimeInstance().format(new java.util.Date());
  String message = "Membership since "+date;
  LOG.debug("Message is = "+message);
  // Get the MD5 Hash of the message
  messageBytes = getMD5Hash(message);
  System.out.println("MESSAGE =>"+new String(messageBytes));
  // Next step is to sign the message
  // For this we use the RSA Algorithm
  try{
    rsaKey = (RSAKey)KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
➥                      KeyBuilder.LENGTH_RSA_512,false);
```

LISTING 8.4 Continued

```
    JxtaCrypto suite = new JxtaCryptoSuite( JxtaCrypto.PROFILE_RSA_SHA1
                                          , rsaKey
                                          , Signature.ALG_RSA_SHA_PKCS1
                                          , (byte)0);
    RSA rsaAlgorithm = new RSA(rsaKey);
    LOG.debug("Setting Public Key");
    // This method computes a public key
    rsaAlgorithm.setPublicKey();
    LOG.debug("Setting Private Key");
    // This method computes a private key
    rsaAlgorithm.setPrivateKey();
    //Get the data for both public and private keys
    RSAPublickeyData publicKeyData
                    =(RSAPublickeyData)rsaAlgorithm.getPublickey();
    RSAPrivatekeyData privateKeyData
                    =(RSAPrivatekeyData)rsaAlgorithm.getPrivatekey();
    // Get a signature Object for the actual signing
    signature = suite.getJxtaSignature();
    signature.init(Signature.MODE_SIGN);
    signatureArray = signature.sign(messageBytes, 0, messageBytes.length);
  } catch (CryptoException cryptoException){
    LOG.error("EXCEPTION IN SIGNING MESSAGE",cryptoException);
  } catch (Exception exception){
    LOG.error("EXCEPTION IN SIGNING MESSAGE",exception);
  }
}
```

First, the `KeyBuilder` is used to create an `RSAKey`. Then you create a suite with the profile `PROFILE_RSA_SHA1`. The `setPublicKey` and `setPrivateKey` methods are called to compute the public and private keys, respectively.

After this is done, a `Signature` object is obtained from the suite. Because in this case you are interested in signing the particular document, you initialize the signature to a `MODE_SIGN` mode. The actually signing is done by the `sign` method.

Verifying a digital signature is also very similar to the signing process, except for the fact that the `Signature` object is initialized to a verify mode (`MODE_VERIFY`).

After this is done, the `verify` method of the signature `Object` can be invoked for the actual verification. This is done by the `init` method of the `SecurityDemo` class.

Trust in P2P systems

As you have seen, JXTA provides various low-level primitives necessary to build secure P2P applications. Nevertheless, they can act only as the building blocks of your security infrastructure. It is up to you to combine these primitives to arrive at a meaningful and preferably powerful security framework for your application, depending on its requirements.

There are many tricky issues in P2P security that you might want to consider. You may recall, from our discussions on Peer Group Management in Chapter 6, the difficulties in managing P2P networks. Security concerns are no different from management concerns because security can also be considered as a integral part of management.

In the client/server world, trust has been a simple issue. People usually trust a centralized authority, such as Verisign. This will also make them trust all other entities certified by Verisign. Thus, a whole trust hierarchy can be built with each trusted entity certifying that another one can be trusted. In this example, Verisign would form the base of this hierarchy.

One problem that is hard to manage is the hierarchy and transitivity of trust. For example, in the hierachial view, you have one central authority that is not good (that is, cannot be trusted) or as with transitivity, the trust is not consistent. For example, even if we are friends, a friend of mine is not necessary a friend of yours.

Trust is a concept that is handled differently in P2P due to its lack of centralization. Because there is no central trusted authority to begin with, the whole concept of trust used in client/server networks becomes meaningless in P2P.

Many interesting approaches have been formulated to approach this issue. One approach is for each peer to rate another peer based on its interactions with it. Ratings can be signed so that they can be authenticated if required. Thus, every peer builds a *trust portfolio*, which can be analyzed at any time and the overall trust evaluated.

Poblano: A Distributed Trust Model

Poblano is a distributed trust model proposed for the JXTA platform. While Poblano is still in its emerging stages at the time of this writing, it holds a lot of promise to be a base framework for trust in JXTA applications.

Poblano divides trust into three components:

- Trust between different peers termed *PeerConfidence*
- Trust between a peer and a codat termed *CodatConfidence*
- The risk factor termed *Risk*

Poblano addresses the issue of calculating the overall trust based on these factors. It also suggests methods for processing, propagating, and updating trust relationships.

As this book went to press, the Poblano system was still being implemented. Please check the security project at `http://security.jxta.org/` for the most up-to-date information.

P2P Security Models

The best ways to come up with a suitable security model is to analyze what your requirements are. We can start by learning from goals, failures, and successes of other applications. Let us now investigate two of the most popular P2P applications, Freenet and Napster. We will not be going into the finer details of these implementations, but will look at them from a broader security perspective.

Freenet

The idea behind Freenet first appeared in Ian Clarke's research paper "A Distributed Decentralized Information Storage and Retrieval System" in 1999. Freenet, as we know today, is an open-source realization of this idea. It is a truly decentralized information publication system.

The Freenet Philosophy

Freenet is built around one core value—freedom of speech. The objective is to make Freenet a channel for free communication that is not moderated by an external entity. Freenet believes that freedom of speech is not possible without anonymity. Also, information must be available for those who seek it and must be survivable from any forms of external control.

The Freenet architecture reflects the Freenet philosophy. The design of the Freenet system is centralized around two goals—privacy and availability.

- Privacy applies to the publisher of the information (one who inserts some content into the network) as well as the consumer of the information (one who retrieves that content from the network).
- Availability means that the information must be present on multiple and possibly unrelated nodes, so that attackers cannot target particular points in the network and make a particular content unavailable.

How Freenet Works

Any Freenet user can insert content of his or her choice into the network identified by a key, which can also be used to retrieve the data from the network. Simple as it seems, let us delve a bit deeper into how this is actually done.

Every node can accumulate data to a limit that can be configured by the user running the node. It also has a routing table to point to other nodes, based on what they are believed to contain. Keys are internally used to represent content. Therefore, the routing table is a mapping between keys and nodes.

When a user tries to insert data into a node, the node verifies if it has any content matching the key of this insert. If the key is not found, the insert request is propagated to the node that has the *nearest* key to the requested key. *Nearness* is calculated based on a predefined algorithm. This goes on until the limit specified by a hops-to-live value of the insert request. If any key clash occurs in this process, the node responds back to the requested node along with the content represented by that requested key and available at that node. If no key clash occurs, a success message is sent to the inserter's node. This is the green signal for the actual data insertion. Note that until this point, the content was never transferred. It was only the key of the content that was propagated. Only after receiving the success message is the actual data for insertion sent.

To retrieve any content from Freenet, you have to know its key. This can be done by either obtaining the key from previously publicized sources or by calculation (as you will see later). After this key is available, the user can request his or her node to retrieve the content.

The node verifies if it has a copy of this content. If it doesn't, it looks up its routing table and passes on the request to the node that has the nearest key matching the requested key. This process continues until the hops-to-live limit is hit. If a match is not found during this process, a failure message is propagated back to the requesting node. If a match is found, the content itself is passed back the same way the request came, but now each and every node stores a local copy of the content before passing it back to its requestor. It also updates its routing table to point to the original node that had the content.

Freenet Security

The security model in Freenet assumes that any node in the network can be potentially hostile, trying to eavesdrop or attack the network. Data exchange is designed to take place effectively under such intimidating conditions.

Before proceeding any further, let us first look at Freenet keys.

Keys

Freenet uses keys to represent content. Three types of keys exist in Freenet:

- A *Keyword Signed Key (KSK)* is the simplest, and is obtained from a descriptive string that the inserter assigns for the content. But it tends to be inserter agnostic; in other words, it is not possible to identify the user who inserted the content.

- A *Signed Subspace Key* (*SSK*) is used to ensure the authenticity of the author. It is generated based on the user's public key as well as a descriptive string.
- A *Content Hash Key* (*CHK*) is used in combination with the SSK as a mechanism for content revisions.

Security at the Protocol Level

All Freenet messages contain three elements:

- A transaction ID
- A hops-to-live value
- A depth value

A Freenet transaction spans end-to-end any process of requesting content or storing content. The transaction ID is randomly generated and can be considered unique for a transaction. The hops-to-live is very similar to a time-to-live (TTL) value found in IP datagrams, and the intent is to limit the lifetime of the message. Every node decrements this value before it passes the message to the next one. This ensures that endless loops do not exist. The depth value determines how deep the message has traveled to reach a particular node. Every node increases this value as it passes along the message. The intention is that when any node needs to reply to the originator with either a success or a failure, it can know what hops-to-live value to set.

Now for the interesting stuff. Even though the time-to-live value becomes 1, a node may choose to forward the message further. Of course, this is with a limited probability. Thus, the message may propagate to a few more nodes until probability works against it and it dies. The primary intention is to puzzle any casual eavesdropper trying to make some analysis based on the hop-to-live value.

For the same reasons, the initial depth is set to a small but random number so that the exact location of the requestor becomes difficult to determine.

Other Security Measures

Another possible security concern comes from the maintenance of the dynamic routing table by each node. As we have seen previously, when a node is involved in routing some requested content back to the requestor, it adds a pointer to the original source of the content into its routing table. Freenet recognizes that this can lead to a security issue, so it allows any node en-route to change this pointer to any other randomly-chosen node, including itself.

Passive Attacks on Freenet

While Freenet is designed to provide anonymity using obscurity, there are chances that the network can be eavesdropped. This can happen when an attacker compromises a large number of Freenet nodes, probably in a particular geography, enabling him to perform some advanced traffic analysis.

Active Attacks on Freenet

KSKs can be subject to dictionary attacks because they are just hashes of descriptive texts, unlike SSKs that involve digital signatures. This is a known vulnerability. It provides one more reason to use SSKs instead of KSKs.

Denial-of-service attacks are possible, although their effectiveness may vary. First, an attacker can flood the network from a node by inserting junk files with randomly generated keys. While key clashes may occur during this process and prevent the attacker from the insertion, it is possible to accomplish this. The attacker may also manipulate the depth of the insert by changing the hops-to-live value to reduce the probability of these collisions. But the limitations to this are that only a few nodes in the vicinity of the attacker's node will store this junk content.

Although requests are a way to replicate data through the network, any request from a node will make the data be stored nearer to that node. Consequently, any subsequent request for that data will not cause any replication. To be effective, the attacker will have to compromise a large number of nodes and issue insert and retrieval commands from multiple random nodes. Such a distributed denial-of-service attack is possible.

Napster

Another good example of security and trust issues can be seen on Napster. In a typical MP3 file transfer in the Napster network, any user who runs the Napster client initially connects to the Napster Index Server. The function of the index server is to maintain the list of users online at any point of time and the files shared by each user. The presence of this index server adds a bit of centralization to Napster.

When a user wants to search for a particular MP3 file, the Napster client contacts the index server. The search is run on the index server. The index server reports the list of files shared, along with identifiers to the other users sharing them. The role of the index server ends here. When the user confirms download for a particular file, the client contacts the target machine directly and downloads the file.

Security Issues in Napster

The main concern of the designers of Napster was to make it unfeasible for Napster to be used as a channel for trading possibly infected files and executables. To enforce this, all files are checked to be sure they are valid MP3 files by looking at the file headers. Only files with legal MP3 headers can be exchanged. Nevertheless, they were proved wrong with the coming of Wrapster.

Wrapster is an application that can be used to wrap any file (document, executable, and so on) into an MP3 format. This can fool Napster into believing that the file is indeed a genuine MP3 and allow it to be exchanged in the Napster network.

Wrapped files can be easily identified. They have a bit rate (a measure of sound quality usually 128Kbps) of 32Kbps and a frequency (number of sound samples per second typically 44,000Hz) of 32,000Hz. Searches can be effortlessly performed using these values to find wrapped files in the network.

Is this a serious issue? Not really, every file that is wrapped at one end has to be unwrapped at the other. This means that the other end needs a Wrapster client too. Without unwrapping, the files are unusable. Trying to execute such a file will invoke the default music player of the operating system, just like any other MP3 file would, and this will fail.

Moreover, users who are well informed enough to know how to wrap and unwrap files to disguise miscellaneous files as MP3s are seldom careless about potentially malicious files.

Even though Wrapster does not impose serious security concerns in Napster, it does provide an example of how a system can be exploited to do what it was not meant to do. Not only must your security design be good, how you implement it is crucial.

Summary

In this chapter, we have had a look at various security concerns that can arise in a P2P network. We have also looked at how JXTA addresses the security issue, both at a platform level and by providing the cryptographic toolkit that can be used by applications built over it. The Cryptographic Toolkit provides a lot of useful APIs, and examples of their use were presented.

Finally, we analyzed two popular P2P networks from a security perspective with the intent of providing valuable insight on various issues related to peer-to-peer security.

In the next chapter, we will be transmitting data to synchronize it between peers. Although we have not added this chapter's security features, because the data is of a personal of group private nature, it is a prime suspect for implementing encryption. Read on and, as an extra thought, think of where and how you would add security in the following implementation.