Core Security Services and Bootstrapping in the Cherubim Security System

Charles Willis

cfwillis@uiuc.edu

Technical Report
2 July 1998

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Ave
Urbana IL 61801
Abstract

This report describes the implementation of the core security services and bootstrapping architecture in the *Cherubim* Security System. The core security services encapsulate the basic facilities required for the *Cherubim* dynamic security policy research project, including encryption, digital signatures, and authentication. The core security services provide a uniform interface to an implementation that can be built with a variety of standard security components. The secure bootstrapping process allows *Cherubim* clients, using a smartcard, to have universal remote access to standard services. This allows us to build a security system with maximum configurability and extensibility which is essential to many emerging applications like active networks and mobile computing.
# Table of Contents

Abstract ................................................................................................................................................. 2
Table of Contents ................................................................................................................................... 3
Introduction ........................................................................................................................................... 4
Core Security Services ......................................................................................................................... 4
Why Build Yet Another Cryptographic API? ....................................................................................... 4
The Heart of the System ....................................................................................................................... 4
  Encryption ......................................................................................................................................... 4
  Decryption ....................................................................................................................................... 5
  Digital Signatures ............................................................................................................................ 5
  Signature Verification ....................................................................................................................... 5
  Secure Hashing .............................................................................................................................. 6
  Secret Key generation from a Passphrase .......................................................................................... 6
Authentication and Key Negotiation ...................................................................................................... 7
  Sequence of Events ......................................................................................................................... 7
  Diffie-Hellman exchange parameters .............................................................................................. 8
Cherubim Bootstrapping Mechanism .................................................................................................. 9
  The Classloader Hierarchy ............................................................................................................... 9
Implementation ................................................................................................................................... 10
  Initial Booting ............................................................................................................................. 10
  Authentication and Key Negotiation ............................................................................................... 11
  Requesting Classes ...................................................................................................................... 11
  The Class Response .................................................................................................................... 12
  Home Server GUI ....................................................................................................................... 12
  Specifics ......................................................................................................................................... 12
System State ....................................................................................................................................... 14
Conclusion .......................................................................................................................................... 15


**Introduction**

This report describes the basic security services upon which the *Cherubim* architecture is built.

The lowest level of the security services is the Core Security Services which abstract the basic cryptographic functions and secret key generation. On top of this base is built higher level functions, namely authentication and secure bootstrapping.

These security services provide an abstraction of the cryptographic functionality on the host computer so that applications and higher level components in the *Cherubim* project will have uniform access to cryptographic functionality.

**Core Security Services**

The Core Security Services consist of the basic cryptographic functions that are used to build up higher level functionality and protocols.

**Why Build Yet Another Cryptographic API?**

A new, simple interface was created because the existing interfaces are overly complicated for the type of operations that are needed for the *Cherubim* system. In fact, the *Cherubim* Core Security Services can be built on top of existing, more complicated, cryptographic APIs. The reference implementation of the Core Security Services (the so-called Generic Core Security Services) is built on top of the Java Cryptographic API developed by Sun Microsystems, as implemented by IAIK.

One important specification is the format for specifying algorithms to be used for the core operations. Algorithms in the *Cherubim* system are to be specified in the same format as in the Java Cryptography Architecture API Specification & Reference available online at [http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html](http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html).

**The Heart of the System**

At the very heart of the Core Security Services are the basic cryptographic operations:

**Encryption**

This is the process of encoding data to protect it from eavesdroppers in the network. Encryption in *Cherubim* can be done with a variety of algorithms (such as IDEA or DES), modes (such as ECB - electronic code book, CFB – cipher feedback, or CBC –
cipher block chaining), and padding schemes (such as PKCS#5 or PKCS#7). The
algorithm name is not specified as an argument to this method or the decrypt method
because it retrieved by calling key.getAlgorithm().

```
byte[] encrypt(byte[] data, Key key) throws
    NoSuchAlgorithmException,
    KeyException,
    IllegalBlockSizeException,
    BadPaddingException,
    NoSuchPaddingException;
```

**Decryption**

This is the complement of encryption; that is, the process of decoding data after it has
been encrypted and transmitted over the network. Decryption in *Cherubim* can be done
by any of the algorithms that can be used for encryption.

```
byte[] decrypt(byte[] data, Key key) throws
    NoSuchAlgorithmException,
    KeyException,
    IllegalBlockSizeException,
    BadPaddingException,
    NoSuchPaddingException;
```

**Digital Signatures**

Digital signatures are the electronic equivalent of a person’s written signature. When
someone receives a document or message with a person’s signature, he or she can be sure
that it really came from that person. Digital Signatures can be done in algorithms like
RSA, DSA, or El Gamal.

```
byte[] sign(byte[] data, PrivateKey key, String algorithm) throws
    NoSuchAlgorithmException,
    InvalidKeyException,
    SignatureException;
```

**Signature Verification**

Signature verification is the electronic equivalent of comparing a person’s written
signature to a reference copy. If a person’s signature on a document verifies, the receiver
is assured that the person signed the document. Like a good detective, the *Cherubim*
system will detect if an adversary transfers the signature from one document to another.
Signatures created using any of the algorithms above can be verified.
boolean verify(byte[] data, byte[] signature,  
    PublicKey key, String algorithm) throws  
    NoSuchAlgorithmException,  
    InvalidKeyException,  
    SignatureException;

**Secure Hashing**

Hashing is the process of transforming a document of any length into a (hopefully) unique number that is relatively small, typically 128-256 bits. It is desirable for a secure hash function to have the property that it is nearly impossible to find two documents that hash to the same value. The *Cherubim* system supports several hash algorithms including SHA-1 and MD5.

byte[] hash(byte[] data, String algorithm) throws  
    NoSuchAlgorithmException;

**Secret Key generation from a Passphrase**

One interesting feature of the security services is that ability to generate a secret key from a passphrase that the user enters. This is used in order to encrypt private keys, etc for storage on disk.

SecretKey passphraseToSecretKey(String passphrase, String  
    algorithm, int keylength,  
    String hashAlgorithm,  
    String characterEncoding)  
    throws  
    NoSuchAlgorithmException,  
    UnsupportedEncodingException;

The basic idea is simple, hash the passphrase using the specified character encoding and hash algorithm. Then, copy hashed passphrase into the rawkey. If the hashed secret is too short it will be repeated to fill the rawkey. If the hashed secret is too long, the last blocks will get xored with the first blocks (this may wrap around several times, that is okay).

Following is the algorithm used in simplified Java (for clarity). At the start of this algorithm, the array hashed contains the hashed passphrase and the array rawkey is the size of the desired key, but full of zeros. Keep in mind that ^ is xor and % is modulo.
for(int i = 0;
  i < max(hashed.length, rawkey.length);
  i++) {

  rawkey[i%(rawkey.length)] =
    (hashed[i%(hashed.length)] ^
    rawkey[i%(rawkey.length)]);

**Authentication and Key Negotiation**

Key Negotiation is accomplished using the Diffie-Hellman protocol. This protocol was developed by W. Diffie and M. E. Hellman and published in “New Directions in Cryptography,” *IEEE Transactions on Information Theory*, November 1976. For a good description of this protocol and its variants, consult *Applied Cryptography, Second Edition* by Bruce Schneier, 1996. Authentication is also performed by this protocol because signatures are added to the Diffie-Hellman messages sent in both directions as illustrated below.

**Sequence of Events**

Described here are the implementation details for this authentication and key exchange. This process is also illustrated in Figure 1. The parameters of the exchange (p and g) are hard coded into the system.

1. Client machine sends a SignedDHMessage (a signed Diffie-Hellman message, signed by the Smartcard) to the server. This message contains the clients part of the key exchange (the generator g to the power of the client’s exponent a, modulo the prime p), the destination of the message (to prevent an attacker from rerouting messages), a timestamp (to prevent a replay attack), and the specific algorithm and length of the session key to be created. This message is signed with the user’s private key.
2. Server verifies the signature, timestamp, and destination on the message. The timestamp gives each message a five minute window of validity in order to account for transmission time and clock skew.
3. Server sends a SignedDHMessage to the client with similar information.
4. Client verifies the signature, timestamp, and destination on the message in the same manner as the server did above.
5. Client and Server generate the shared secret as described in the protocol.
6. Client and Server hash shared secret to a secret key. The shared secret is hashed using a secure hash algorithm. SHA-1 is the default, but others like MD5 or MD4 could be used. The final problem is that the output of the hash may not be the same size as the key that is desired. If the secret is too short, it is merely repeated enough times to fill the key. If the hashed secret it too long, the secret is “wrapped around” using XOR.
Finally, the session key expires after one hour. After this time, key negotiation is repeated. This limits the amount of ciphertext that an attacker will have available to break the encryption key. It also strikes a good balance between minimizing the amount of data that could be recovered by an attacker who guesses the key, and maintaining good system performance by avoiding too many key negotiations.

**Diffie-Hellman exchange parameters**

These parameters are hard coded into the system in order to ensure uniformity and prevent a possible attack using weak parameters. The first parameter is a 2048 bit prime number. It is shown below in hexadecimal.

---

**Figure 1 – The Diffie-Hellman Authentication Protocol**
The second parameter is called the generator (or \( g \) for short). For this prime number, choosing an appropriate generator is easy.

\[
g = 2
\]

**Cherubim Bootstrapping Mechanism**

This section describes the secure bootstrapping mechanism used by clients in the Cherubim system. First, the classloader hierarchy is explained, which will illustrate how the bootstrapping concept fits into the overall Cherubim system. Then, the mechanisms that secure the bootstrapping process are described.

The basic idea of the Cherubim bootstrapping system is quite simple. A classloader is used and if a class is needed that is not available locally, the classloader loads it over the network. There are a fair number of details to consider, but this is the general concept.

**The Classloader Hierarchy**

The classes in the Cherubim system are split into three categories that are loaded by different classloaders as described here and illustrated in Figure 2:

**Primordial Classes** - The primordial classes contain the java core classes (packages that begin with java.), as well as the necessary cryptographic code from the Cherubim and other packages. The java core classes are assumed to be present and reliable on the client machine, while the other primordial classes are taken to the client machine by the user on a smartcard or similar device.

**Jurassic Classes** - The jurassic classes consist of those classes present on the user's home machine. This includes the classes that make up the jacob object request broker, the classes that make up any applications that reside on the user's home machine, and the Cherubim policy library (which consists of the basic blocks from which specific policies are constructed).
Active Capability Classes - These classes are loaded by the active capabilities classloader and are the code that makes up the access control specifications in the Cherubim system. More information on active capabilities is available in the other reports on the Cherubim web site.

Implementation

In order to securely load the jurassic classes over the network to the client machine, encryption, authentication, and digital signatures are necessary. This section describes the specifics of how the client and server authenticate one another and agree on a session key, as well as how the client machine loads the jurassic classes from the home machine.

Initial Booting

Initial booting of the Cherubim System proceeds as enumerated here. For each step of the process, we assume certain things from the underlying operating system. We assume that the attacker is another user on the system.

1. Client machine boots its Operating System and Java Virtual Machine (JVM) – the specifics of this process are not considered in our project, but this may be done with the system developed for AEGIS or a similar system. Care must be taken so that another “trojan horse” operating system or JVM is not loaded instead of the desired one. Also, the operating system must ensure that the java core classes (i.e. the file classes.zip) is authentic.
2. User runs Boot program from the smartcard. Here, the operating system must ensure that the correct program is executed.
3. Boot program prompts for passphrase. This requires that the operating system provide a secure channel from the keyboard to the JVM so that other users cannot eavesdrop and learn the passphrase.
4. Boot program hashes passphrase into a symmetric key.
5. Boot program uses symmetric key to decrypt smartcard (which contains the user’s private keys and other information). For this reason, the operating system must load the JVM into a protected area of memory because the user’s passphrase and private keys will be present in the JVM’s memory. Other users must not be able to access the JVM’s memory, and further the operating system must not write memory pages to disk in such a way that other users could subsequently read them.
6. Client machine makes a socket connection to the user's home machine
7. Home machine spawns a Connection thread to communicate with the client
8. Client begins key negotiation with the server

At this point it may seem that we are trusting a lot on the client machine. This is true, the focus of the *Cherubim* system and its bootstrapping architecture is system where the machine (including hardware, operating system, and administration) is trusted, but the network is not. As will be seen below, the *Cherubim* system makes no assumptions about the integrity of the network or the authenticity of any host with which it communicates.

**Authentication and Key Negotiation**

Authentication and Key Negotiation are performed using the Diffie-Hellman protocol as explained above. Once the hosts have authenticated themselves to one another and decided upon a session key, the client can begin loading classes over the network.

**Requesting Classes**

After this authentication has taken place, the Client can then request classes as follows:

<table>
<thead>
<tr>
<th>Packet Data Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Name</td>
</tr>
</tbody>
</table>

**Figure 3 – The Class Request Message Format**

1. JurassicClassLoader receives request for a class on the client
2. JurassicClassLoader checks to see if requested class is in the class cache; if so, return it
3. JurassicClassLoader checks to see if the primordial classloader can load the class (i.e. if the class is in the CLASSPATH); if so, return it
4. JurassicClassLoader checks if existing session key is more than one hour old; if so, negotiate a new one as above
5. JurassicClassloader sends a SEClassRequest (a signed, encrypted class request, signed by the Smartcard and encrypted with the IDEA session key) to the home server over the existing socket
6. Server verifies the signature, timestamp, destination, and sequence number on the SEClassRequest

**The Class Response**

<table>
<thead>
<tr>
<th>Packet Data Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Name</td>
</tr>
</tbody>
</table>

**Encrypted with IDEA Key**

**Figure 4 – The Class Response Message**

1. Server loads the requested class off the local disk, if available
2. Server sends the class in a SEClassResponse (a signed, encrypted class response, encrypted with the IDEA session key) to the client
3. JurassicClassloader verifies the signature, timestamp, destination, and sequence number on the message
4. JurassicClassloader adds the class in the message to the class cache
5. JurassicClassloader returns the class in the message to the process that called it

**Home Server GUI**

Finally, a graphical user interface has been created for the home server side of the bootstrapping process. This interface displays the clients that are connected to this home machine, their state in the key negotiation process, and messages from the server internals to the administrator. By clicking on one of the users connected, another dialog box is displayed with more specific information. A screenshot of this interface is shown in figure 5.

**Specifics**

The bootstrapping system for the *Cherubim* project is highly configurable, with reasonable (perhaps slightly paranoid) defaults for all parameters. It allows the user to choose the encryption, signature, and hash algorithm involved as well as the size of the keys.

- Message Digesting – default is SHA-1, though other algorithms like MD4, MD5, or RIPE-MD 160 could be used.
• Digital Signatures – default is SHA-1/RSA, though other algorithms like DSA or El Gamal could be used.

• Session Keys – default is IDEA, though DES, Triple DES (3DES), Blowfish, or SPEED could be used. System defaults to using Cipher Block Chaining Mode and PKCS#5 padding though this can be changed as well.

• Passphrase hashing – default is to use SHA-1 hash algorithm. The resulting data is converted into a Secret Key as described above.

• Character encoding for hashing passphrase – default is UTF8 because this is a common character encoding that should be present on all systems.

• Symmetric Key for writing to stable storage – again the default here is IDEA in Cipher Block Chaining mode and using PKCS#5 padding, but other algorithms, modes, and padding schemes can be used.

Figure 5 – Screen Shot of the Home Server GUI
• Diffie-Hellman key exchange parameters – these parameters are hard coded in the system as described in the authentication section above.

System State

The system state is in many ways the glue that bind the security services described in this report to the rest of the Cherubim system.

System state is a static object in the Cherubim system. It contains all the information that other parts of the system need regarding the current state of the system. The most important part of the system state is the CherubimPrincipal. This object encapsulates a principal in the Cherubim system. That is, it contains the principal’s unique name, key pair, secure socket layer (SSL) certificate chain, and active capabilities. Also important is the fact that the principal’s private key should not leave the CherubimPrincipal class. This class has methods to sign data. (though currently the key leaves to set up the SSL trust decider this needs to be fixed).

For a user, a Smartcard object is used instead of a CherubimPrincipal object (the former being a subclass of the latter). The extra information contained in the Smartcard is the user’s home agent (and some other information?).

• CherubimPrincipal – object representing the principal

• Active Capabilities – database of active capabilities keyed on the CORBA name of the object

• Core – current implementation of the Core interface

• Network – represents the current state of the network, used by the Cryptographic Policy

• Cryptographic Policy – maps Network configuration to SSL Cipher Suites

• Jurassic Class Loader – this is the Class Loader that is being used to load classes from the home machine. This object also communicates active capabilities to and from the home machine.
Conclusion

This will allow not only secure communications among hosts on the active network (essential for the transportation of active agents or active capabilities), but also allow for secure connections between active routers. This will make a great step toward securing the Cherubim system from malicious users on the network.

Those seeking further information should consult the Cherubim project web site online (http://choices.cs.uiuc.edu/Security/cherubim/) where other papers and reports from this project can be found. Also available is the API documentation for the security services described in this report (http://choices.cs.uiuc.edu/Security/cherubim/core/javadoc/packages.html).