A secure client/server java application programming interface

Tawfik Lachheb

A SECURE CLIENT/SERVER JAVA APPLICATION PROGRAMMING INTERFACE

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Tawfik Lachheb

March 2004

A SECURE CLIENT/SERVER JAVA APPLICATION PROGRAMMING INTERFACE

A Project

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Tawfik Lachheb

March 2004

Approved by:

[redacted]

Dr. Tong Lai Yu, Chair, Computer Science — 3/2/04 Date

[redacted]

Dr. Josephine G. Mendoza

[redacted]

Dr. Richard J. Botting

ABSTRACT

Nowadays, computers constitute a very important part of our modern life; the Internet has transformed today's world to a 'Global Village'. Computers are involved in about every aspect of our life, from e-mail to instant messaging to shopping and banking. An increasing number of people are connecting to the Internet to pay bills, transfer money or trade stocks; this would be impossible without secure computer systems. But the growth of computer systems use is coupled with a growth in computer crime opportunities. Computer applications must run in a secure environment; they should prevent unauthorized people from accessing private data. It must be infeasible for a hacker to withdraw money from someone else's bank account and an unauthorized stock trader must be unable to deny buying or selling shares. Secure systems are designed so that the cost in money or time of breaking any component of the system outweighs the rewards; in other words, the security of a system should be proportional to the resources it protects.

Secure computer systems must ensure confidentiality; secret data exchanged between different components need to be encrypted to ensure that data will not be modified in transit even if the data were snooped by a hacker during transit. These systems should also prevent unauthorized subjects from discovering private information on a host computer. Secure systems must use authentication to make sure that the sender is really who he or she is claiming to be and make it possible to know, when needed, the identity of parties involved; user authentication can also provide non-repudiation if a digital signature is used.

The purpose of this project is to develop a generic Java Application Programming Interface (API) that allows applications to provide secure functionalities such as data transfer, key management and digital signature etc. The API is easy to use and encapsulates all security operations so that a developer does not need to worry about its inner working. It exposes simple methods; a user needs to know very little about computer security to use it. The API contains two parts: a server side and a client side. The server side manages users and user keys; the client side includes encryption and decryption capabilities as well as methods to communicate with the server side. The project also provides a sample online E-mail application that uses this API. The E-mail

application contains a friendly web interface for the users to send, receive E-mails and manage their E-mail accounts in a secure manner; it also allows users to manage public keys belonging to their correspondents. The server side of the E-mail application manages user E-mail accounts and the communication with mail servers for sending and receiving E-mails.

The Java secure API was developed to work in any environment capable of running a Java Virtual Machine (JVM) version 1.4.2 or higher; the sample E-mail application is intended to work within Microsoft Internet Explorer browsers version 6.0 and above or Netscape Navigator version 7.0 and above. We will assume that the client machines have browsers with the Java Plug-in version 1.4.2 or higher installed to run the E-mail client application.

The API was fully validated with included test programs. On the client side, individual algorithms are tested for integrity with other encryption/decryption software, such as PGP 8.0 trial version. The server side was validated using a test client that generates random inputs and verifies the outputs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

Most computer users interact with secure systems such as online banking systems where large amounts of money transferred daily are at stake. But there are many scenarios where security is not a common part of computer systems such as E-mail or instant messaging. The motivation for this project came from the idea of making computer security a more important part of users' experience with computer systems. Some users might want to secure their hard disks, exchange secret E-mails or be able to use instant messaging without fear that someone might be eavesdropping. The Secure API is developed using the Java programming language. The Java language offers the unique advantage of a "Write Once, Run Anywhere" capability. Java programs are written to run on a Java Virtual Machine (JVM); a programmer can develop a program and expect it to run on the JVM of different computers. In the Java programming language, the notion of the Java sandbox makes it possible to ensure that Java programs respect their hosts. By default, programs are prevented from reading privileged files, consuming too many resources or communicating over sockets on behalf of the host computer. Permissions are required to be explicitly granted for the programs to do so. In early Java versions, Java security applied only to applets running within a Java enabled browser under strict security limits. But in the Java 2 Platform, the sandbox security model can apply to both Java applications and Java applets running under Java Plug-in. As a part of the built-in libraries included in the Java Developer Kit (JDK), a default security API implementation is available to allow encryption, digital signature and other security related functionalities. The Java security model is designed to allow different implementations of the security API [5]; it is implemented as a set of abstract Java interfaces. The implementation of these interfaces, also known as providers, can be plugged in seamlessly for use with any application. Developers are able to select different security providers for their applications. Java 2, version 1.4.2 comes with two security providers: one implements DSA-based algorithms and one implements RSA-based algorithms for encryption [1,6]. It also comes with two other security providers: one with JCE and one with JSSE [1,5]. The secure API developed in this project utilizes an implementation of the Pretty Good Privacy (PGP) which is not part of Java's security extension. PGP is a hybrid cryptosystem that combines some of the best features of both conventional and public key encryption. When a user encrypts text with PGP, the text is first compressed in ZIP format then encrypted using a one-time session key generated from random mouse movements or keystrokes from the user; finally the session key is encrypted using the recipient's public key and transmitted along with the ciphertext. To decrypt a message, PGP recovers the session key using the user's private key and decrypts the ciphertext using the session key. The Secure Client/Server API we developed provides a simple way to implement solutions for users seeking more security in their computer systems. The API will allow developers to focus on specific aspects of their applications rather than design and implement the security features of their products.

Purpose of the Project

The purpose of this project is to implement a generic Java API that allows application developers to easily incorporate security functionalities into their applications. Currently, development of security functions using the Java language requires good knowledge of the Java security model. Furthermore, Java's security extension does not provide support for the PGP security protocol that has proven to be very secure and efficient. The Java Secure API we developed provides a PGP implementation of security features such as encryption, decryption, key management and digital signature. The API can be used to secure any Java application with minimal effort. The API is easy to use and hides all the details of security operations that are irrelevant to the user. The project also provides a sample online E-mail application that uses this API; this E-mail application presents a good reference for the usage of this secure API.

Project Products

This project delivered the following:

API Source Code and Compiled Classes

The source files contain the implementation of all the security methods provided by the API as well as comments within the source for relevant statements and methods. We also deliver the compiled classes in a Java

Archive (JAR) file that developers can easily include in their projects.

<u>API User Guide</u>

The user guide contains documentation of this project's products. Detailed instructions are provided for:

- Using security functions.
- Using the API client.
- Deploying the API server.
- Using the E-mail client.
- Deploying the E-mail application on a server.

The user guide includes sample code for reference as well as the JavaDoc for all classes in the API.

<u>E-mail Sample Application</u>

To illustrate the usage of the API we deliver a sample online E-mail application that developers can adapt to their project along with reference to the user guide. The secure E-mail client consists of a web interface implemented as a Java applet using the client side of the API; the server side is implemented as a Java servlet that processes requests originating from the client application.

CHAPTER TWO

REQUIREMENTS AND SPECIFICATION

Project Components

The mission of this project includes the design and implementation of a security client/server API. This API provides classes and methods to perform security operations such as encryption, decryption and digital signature. The project can be divided into three components: the API client side referred to as the security client, the API server side referred to as the key manager service and a sample online secure E-mail application; this sample application demonstrates the use of the secure API. The Borland JBuilder Version 8 personal edition is the coding platform used to implement this project. JBuilder is a cross-platform environment for building software; it improves productivity of developers thanks to features like integrated JSP/Servlet support, integrated tools for database development and support for many version control systems.

Security Client

The security client consists of a library of Java classes that can be used by an application to perform security operations and interact with the key manager service. The security client provides the following functionalities:

- Public key encryption/decryption methods: these methods take a plaintext or a ciphertext and a key as inputs and return the encrypted/decrypted data. Typically, the encryption/decryption key would be retrieved from the server using key manager service, but a key unknown to the server can also be used. PGP private keys are encrypted using a secret passphrase known only by the key's owner; hence, the user's passphrase is required to recover the private key for decryption. Two public key encryption algorithms are supported: RSA and ElGamal. RSA was developed by Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman in 1977; it uses exponentiation modulo a product of two large prime numbers to encrypt and decrypt (See chapter 4). Elgamal was developed by Taher ElGamal and is based on the discrete logarithm problem [2].
- Conventional encryption/decryption methods: these methods take a plaintext or a ciphertext and a key

as inputs and return the encrypted/decrypted data. For confidentiality reasons, secret keys are not saved on the server; they are managed by the security client. The following symmetric encryption algorithms are supported: IDEA, DES and TripleDES. These algorithms have proven to offer the best trade-off between speed and security.

- Digital signature methods: these methods generate a digital signature of user data using a provided private key. The private key used can be downloaded from the server or provided by the security client. The digital signature algorithms supported are: MD5WithRSA, SHA1withRSA and DSA. MD5WithRSA and SHA1withRSA combine RSA with the strongest message digest algorithms we support.
- Message digest methods: these methods are used to generate the message digest of a user data based on a selected message digest algorithm. Message digest algorithms supported are: MD2, MD4, MD5, SHA, SHA0 and SHA1. These algorithms are the most commonly used and thus are more likely to be needed from the API we develop.

- Methods to communicate with the key manager service: these methods provide an interface to the key manager service. A client application is able to publish a key pair on a server or retrieve a user's public key etc.

Key Manager Service

The key manager server is implemented as a Web service accepting requests through a Simple Object Access Protocol (SOAP) interface [15]. The key manager service is developed in a way that allows plugging in custom implementations of the service interface. This project provides an implementation that manages server information in a Relational Database Management System (RDBMS); other implementations can proceed differently but keep the interface unchanged from the security client's point of view. For example, the public keys could be stored in a remote key server. The key manager service provides the security client with the following services:

- User management: the key manager service holds and provides information about users having their keys published on the server.

- Key management: client applications are able to publish public keys or PGP key pairs and get other information on a public key. The security client can request a list of users each of whom has signed a certain key or information about the owner of a public key.
- User trust: the server keeps trust information between pairs of users. The trust level of a user represents the level of legitimacy of public keys he or she introduces. The security client has the ability to retrieve the trust level of one user to another. The security client can also request the key manager service to compute the legitimacy level of a user's public key based on all trust information available on the server.

Sample Application

The sample application consists of a secure online E-mail management tool. It allows sending and receiving encrypted and unencrypted E-mails as well as publishing public keys to the server. The client E-mail application is implemented as a Java applet running on the Java plug-in version 1.4.2 or higher; the server side of this application is developed as a Java servlet running on the Sun One Application Server version 7. Information related to this application is stored in a MySQL version 4.0.15 RDBMS. Database connections are managed by a connection pool provided by the application server. The E-mail servlet handles communication with the mail servers using the JavaMail API. The client E-mail application includes checking for E-mails, reading E-mails, sending E-mails, as well as the communication with the key manager service for submitting keys to the server, signing keys and setting trust values between users.

Validation Criteria

Prior to being put into practice, all parts of the project were verified to meet all the requirements stated previously in this section. The following validation tests were performed:

- Unit Tests: test the individual method implementations to ensure that they perform as desired. The unit tests were performed using the JUnit testing framework.

- Integration tests: test the interfacing of different components of the system after they are assembled. These tests ensure that the system elements were properly integrated and will perform as expected.

CHAPTER THREE
PROJECT APPROACH

Introduction

The project consists of the development of a secure client/server API and a sample application that illustrates its usage. The server side of the API is exposed as a Simple Object Access Protocol (SOAP) web service that allows plugging-in custom implementations of the web service interface. A default implementation is provided in this project; it utilizes a database and standard SQL to manage user keys and other data. In order to efficiently implement the interaction with the database, the Java Database Connectivity (JDBC) is used. JDBC-based programs are written once to run on any platform connecting any local or remote relational database; this reduces maintenance overhead and improves flexibility [16]. The client side of the secure API contains security methods such as encryption and digital signature as well as the ability to communicate with the key manager service through SOAP client stubs. The sample application consists of a web e-mail application with a user interface implemented using HTML, JSP, applet and JavaScript technologies. The server side is developed as a Java servlet that manages users' E-mails and account information using a database. Similar to the key manager service, the servlet implementation is database independent. Figure 1 illustrates the structure and relationships between the components of this project.

Security Client Design and Implementation

The security client functionality can be decomposed into two types of methods: security methods such as encryption/decryption and methods for communication with the key manager service interface. In the design phase of the security client, we start by identifying the security methods to be exposed; the methods are then grouped based on the type of security operations they perform; for example, all message digest methods using different algorithms are grouped into a single package. The interface for communication with the key manager service is designed simultaneously with the design of the key manager service; to each method in the key manager service corresponds one method in the security client used to invoke the service. Finally, once all the

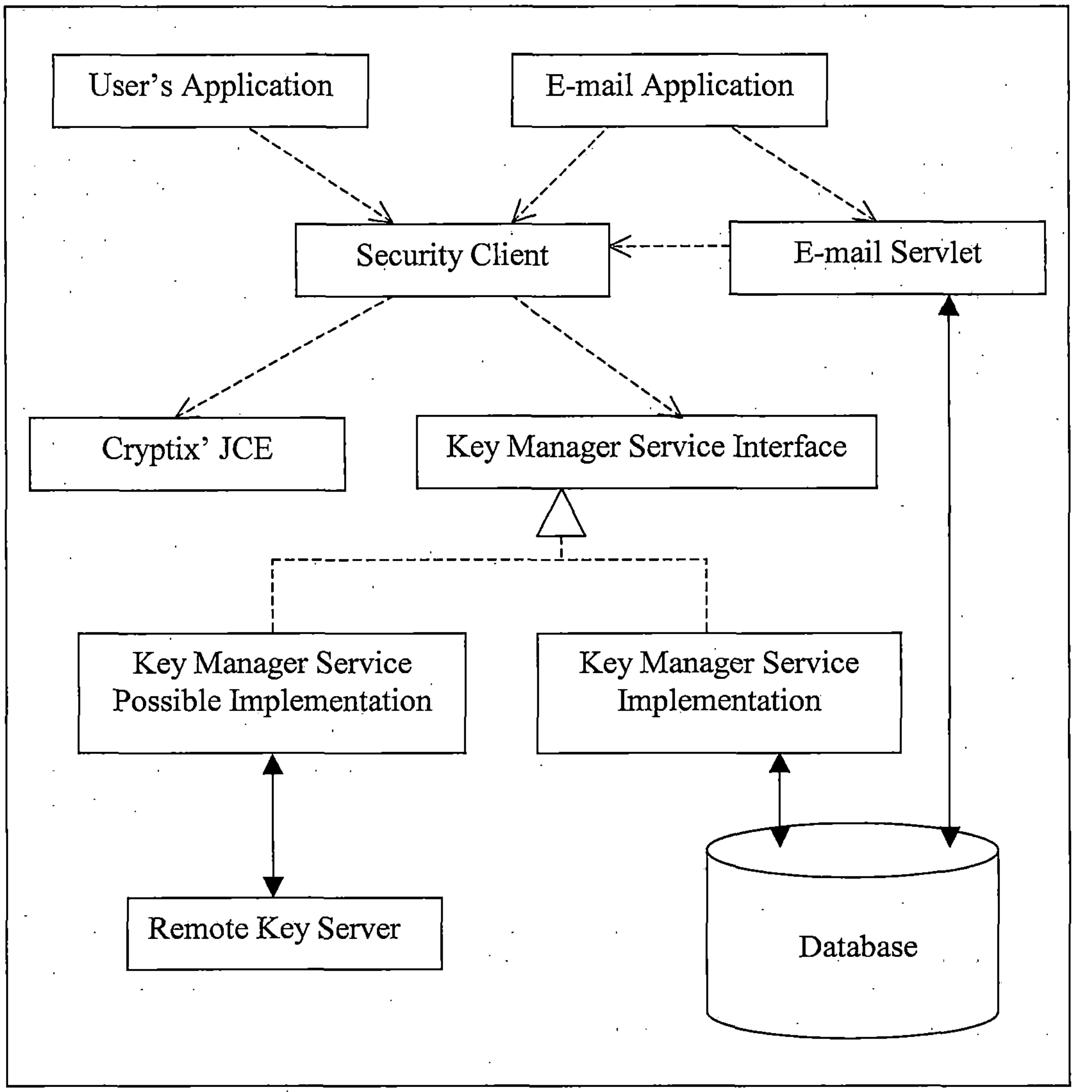

User's Application
E-mail Application
Security Client
E-mail Servlet
Cryptix' JCE
Key Manager Service Interface
Key Manager Service Possible Implementation
Key Manager Service Implementation
Remote Key Server
Database

Figure 1. Overall View of the Project Components

methods are implemented, the security client is validated using classes of unit and integrity tests.

Key Manager Service Design and Implementation

The key manager service design phase starts by defining the interface for the service. The methods exposed provide the means to manage users' information, publish and retrieve public and private keys. The implementation we provide manages keys and users' information using a database. Since the key manager service publishes a Java interface, the security client is independent of the service implementation; the security client can perform unchanged when the service implementation is changed. The entity-relation (E-R) model is designed before proceeding to the service implementation. The E-R Model contains a concise description of the entities involved in a system, relationships and required constraints. The E-R model is visually represented by an E-R diagram. The next step consists of normalizing the E-R model. Normalization is the process of decomposing the system's entities and relationships by breaking up their attributes into smaller entities and relationships that possess desirable properties; this is to ensure that insert, update and delete anomalies do not occur in the system's database over time [16, 19]. The goals of the normalization include minimizing redundant data, reducing inconsistent data and designing systems to be easier to maintain.

The security manager service is then developed to run as a web service that fulfills requests from the security clients via SOAP messages. Finally the security manager service is validated using classes of tests.

E-mail Application Design and Implementation

The sample E-mail application is implemented in two components: a web client that runs as a Java applet within a browser and a Java servlet that runs on a servlet container and receives requests from the applet. The applet communicates with the servlet by sending and receiving messages using the HTTP protocol. The servlet uses a database to store information. The E-R model is defined before implementing the servlet.

CHAPTER FOUR

SECURITY CLIENT DESIGN AND IMPLEMENTATION

Conventional Encryption

Until the development of the public-key encryption, there was only one type of encryption involving the use of a single key: conventional encryption, also referred to as symmetric encryption [10]. The encryption process consists of an algorithm and a key; it converts an original message known as plaintext to an apparently random message referred to as ciphertext. Once produced, the ciphertext is transmitted then decrypted at the destination using the same encryption key (See Figure 2).

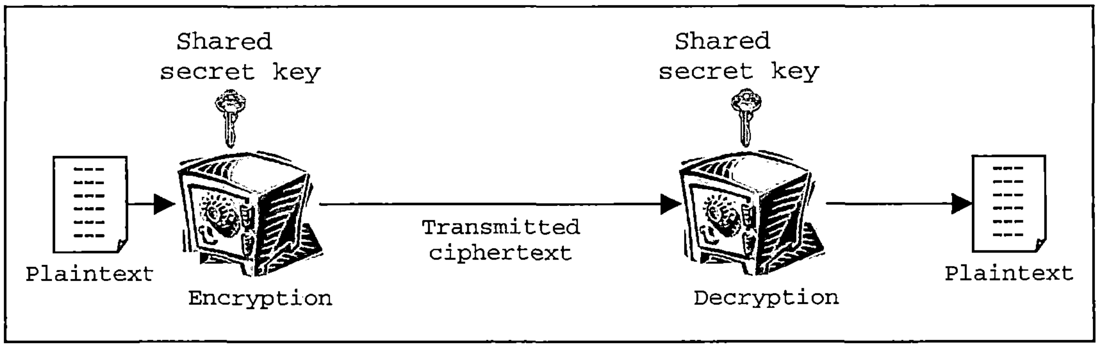


Figure 2. Symmetric Encryption

There are two characteristics of the ciphertext that can be used to deter statistical cryptanalysis: diffusion and confusion [2]. In diffusion, the statistical structure of the plaintext is dissipated into large portions of the ciphertext whereas confusion consists of making the relationship between the statistics of the ciphertext and the encryption key as complex as possible.

The encryption process can be seen as a function parameterized by the encryption key. Therefore, if we consider a plaintext $X = [X_1, X_2, \dots X_m]$ where the m elements are in a finite alphabet, a key $K = [K_1, K_2, \dots K_m]$ and a ciphertext $Y = [Y_1, Y_2, \dots Y_n]$ then the encryption can be written as:

$$Y = E_k(X) \quad (1)$$

The decryption process can be represented by:

$$X = D_k(Y) \quad (2)$$

The earliest known use of a cipher was by Julius Caesar [7]. The Caesar cipher is a substitution cipher where each letter is replaced by the third letter further in the alphabet. In general, a substitution cipher consists of shifting the alphabet k times; this can be represented by:

$$Y_i = X_i + k \bmod(26) \quad \text{for } i = 0 \dots m$$

and $$X_i = Y_i - k \bmod(26) \quad \text{for } i = 0 \dots m$$

In the 19th century, Lewis Carroll developed a substitution cipher where letters in the plaintext are shifted differently based on their location. Given a key $k = (K_1, K_2, \ldots, K_m)$ and a plaintext $X = (X_1, X_2, \ldots, X_n)$, the ciphertext Y is computed by encrypting X in blocks of m letters. A letter Y_i in a block of the ciphertext is computed as $Y_i = X_i + k_i \bmod(26)$. This cipher is harder to break than the Caesar cipher especially as the size of the key increases.

The Hill cipher, developed by Lester Hill in 1929, is an improvement of the substitution ciphers [7]. It consists of taking m consecutive letters frokm the plaintext at a time and replacing them with m ciphertext letters. It is represented by m linear equations; for m = 3, the cipher can be defined by:

$$C = KP \pmod{26}$$

where $P = (P_1, P_2, P_3)$ is a vector of 3 letters from the original plaintext that is transformed into a new vector of three letters $C = (C_1, C_2, C_3)$ using the key K represented by a 3X3 matrix:

$$\begin{pmatrix} C1 \\ C2 \\ C3 \end{pmatrix} = \begin{pmatrix} k11 & k12 & k13 \\ k21 & k22 & k23 \\ k31 & k32 & k33 \end{pmatrix} \begin{pmatrix} P1 \\ P2 \\ P3 \end{pmatrix} \pmod{26}$$

Decrypting the message requires computing the inverse K^{-1} of the matrix K defined by $KK^{-1} = K^{-1}K = I$, where I is the identity matrix. In this cipher, diffusion is achieved by using three letters of the plaintext to produce one letter in the ciphertext; in other words, information consisting of a character in the plaintext is diffused in three characters in the ciphertext. A higher level of confusion is achieved by using a linear combination to compute each letter in the ciphertext; this is more complicated than the simple addition of a constant number in a substitution cipher.

More powerful encryption can be achieved by subjecting the plaintext to multiple stages of encryption. Rotor machines are based on this principle. They were used in Germany's Enigma and Japan's Purple machines in the Second World War. In 1938, William Friedman built an identical purple machine to decode Japanese secret messages; it provided decisive intelligence to the United States military that resulted in victories such as the one in battle of Midway. Marian Rejewski determined the wiring of the Enigma's rotors in the winter of 1932, since then, Poland was able to read thousands of German secret messages. In July of 1939, France and Great Britain were delivered replicas of the Enigma machine; this played a major role in the allies' victory in the Second World War [2]. Rotor machines contribute significantly in the Data Encryption Standard (DES), one of the most widely used ciphers today.

Data Encryption Standard

The Data Encryption Standard (DES) algorithm was adopted by the National Bureau of Standards in 1977 and is still widely used. It is a block cipher that uses a 56-bit key where the data is encrypted in 64-bit blocks; each block is subjected to 16 rounds of transformation. Triple-DES is a more secure variation of DES using three keys; given three keys K_1, K_2 and K_3, the cipher C of a plaintext text P is defined by $C = E_{k3}(E_{k2}(E_{k1}(P)))$.

IDEA

The International Data Encryption Algorithm (IDEA) is a symmetric block cipher developed by Xuejia Lai and James Massey from the Swiss Federal Institute of Technology [10]. IDEA uses 128-bit keys to encrypt data in 64-bit blocks.

Blowfish

Blowfish is a conventional bloc cipher developed by Bruce Schneier [10]. It is known for its speed and low memory use. It is simple to implement and permits variable key lengths, which provides a tradeoff between speed and levels of security.

Table 1 summarizes advantages and disadvantages of the symmetric algorithms we cited.

Table 1. Comparaison of Conventional Encryption Algorithms

Algorithm	Publication Year	Advantages	Disadvantages
DES	1975	Well tested	Key too short, slow
Triple-DES	N/A	Very well tested	Three times slower than DES
IDEA	1991	PGP Popularity	Commercial license required
Blowfish	1994	Fast encryption and decryption, popular	Slow key setup

Public Key Encryption

Until the development of public-key cryptography, most cryptographic systems have been based on basic substitution and permutation. The major drawback of conventional encryption is the problem of secret key distribution; it becomes a challenge when dealing with unknown parties. In 1976, Whit Diffie and Marty Hellman from Stanford University published a paper titled "New directions in cryptography" where they described the concept of public key encryption [4]. Public key algorithms are based on mathematical functions instead of substitution and permutation. Public key cryptography consists of an asymmetric process involving the use of two keys as opposed to a single key in conventional encryption as shown in Figure 3. The use of two keys provides major advantages in confidentiality, key distribution and authentication [1].

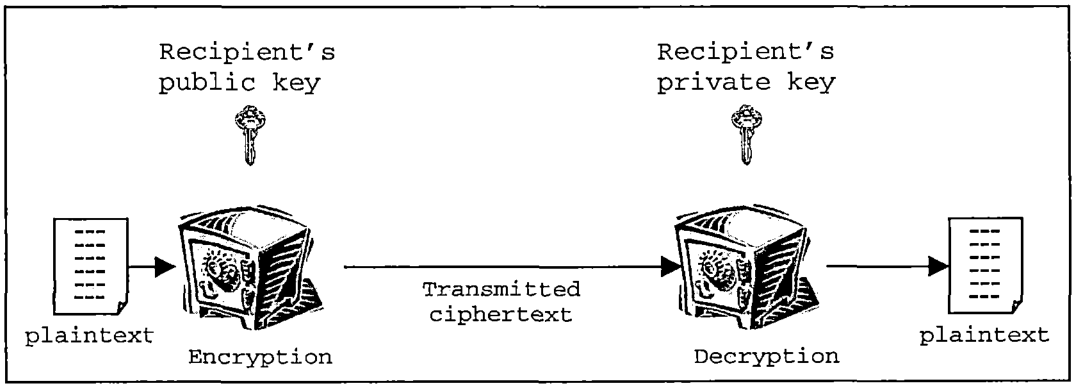


Figure 3. Public Key Encryption

Currently, there is a single well-known implementation of the public key encryption approach that is used: RSA. RSA was developed in 1977 by Ron Rivest, Adi Shamir and Len Adelman putting Diffie and Hellman's concepts into practice [10]. RSA encrypts messages in blocks having a binary value less than a certain predefined value n. For a plaintext block M, encryption and decryption can be expressed by [2]:

$C = M^e \bmod n$ and $M = C^d \bmod n = M^{ed} \bmod n$ where e and n are numbers known to both sender and receiver and d is known only to the sender. In other words, (e,n) represents the public key and (d,n) represents the private key. This is based on the fact that it is possible to find e, d and n such that for all $M < n$, $M^{ed} = M \bmod n$. Based on Euler's theorem, given two prime numbers p and q and two integers n and m where $n = pq$ and $0 < m < n$, we have the following relationship for every integer k: $m^{k\phi(n)+1} = m \bmod n$ where ϕ is the totient function. It is the number of positive integers less than n and relatively prime to n. If p and q are prime then $\phi(pq) = (p-1)(q-1)$. Thus, by taking $d = e^{-1} \bmod \phi(n)$ which is equivalent to $ed = k\phi(n)+1$, we can achieve $M^{ed} = M \bmod n$. To generate a key pair, two prime numbers p and q are chosen, n is taken as pq. Then e is chosen such that $\gcd(\phi(n),e) = 1$. Finally, d is computed as $e^{-1} \bmod \phi(n)$. The strength of RSA is based on the fact that given a value of n, it is very difficult to find the p and q values.

Hash Functions

Usually, a major concern when a message is delivered is to detect whether changes were made to the message after it left its origin. This is done by producing a fingerprint of the message at a time when the message is known to be authentic. The fingerprint is referred to as the message hash. The message hash is calculated at the source and appended to the message. At the destination, the message hash is re-computed; if the same value is produced then the message is considered authentic. To be considered reliable, a hash function H must be [4]:

1. One-way: given y, it must be hard to find x such as $H(x) = y$.
2. Weakly collision-free: given x_1, it must be hard to find x_2 such as $H(x_1) = H(x_2)$.
3. Strongly collision-free: it must be hard to find any pair (x_1,x_2) such as $H(x_1) = H(x_2)$.

One of the most commonly used hash functions is the MD5 algorithm developed by Ron Rivest (the "R" in RSA). MD5 processes messages in 512-bit blocks to produce a 128-bit message digest. Based on the MD4 hash algorithm (precursor of MD5), the National Institute of Standards and Technology developed the Secure Hash Algorithm (SHA);

SHA-1 is a revised version widely used today. SHA-1 has been well tested and validated over time; it is very strong against brute force cryptanalysis attacks but it performs slower than SHA and MD5.

Digital Signature

Unlike all other security tools that protect information from intruders, digital signature protects the two parties exchanging information from each other. As with handwritten signature, digital signature binds a document to its author. It also serves as a proof that the information exchange did happen in a similar way to a witnessed handwritten signature. Without the use of digital signature, a receiver of a transferred fund can increase the amount and claim that the larger amount was authorized for transfer. Another scenario could be of an E-mail instructing a stockbroker for a transaction; the sender later learns that the transaction turned bad and denies sending the E-mail. Digital signature can be achieved using RSA by encrypting the message or a hash of it using the sender's private key. The signature validity depends on the authenticity of the sender's private key. The sender can still claim that his or her private key was lost or stolen.

The Digital Signature Algorithm (DSA) was introduced by David W. Kravitz [7]. In 1994, DSA became the U.S. Federal Information Standard FIPS186 called DSS, making it the first digital signature algorithm adopted by any government. DSA is based on the difficulty of calculating discrete logarithms; the discrete logarithm y of x to the base b modulo m is defined by: $x = b^y \bmod m$. In addition to the user's private key, DSA uses three public parameters. First a 160-bit prime number q is chosen. Next, a prime number q that divides p-1 is chosen of length between 512 and 1024 bits. Finally, an number g of order q modulo p is chosen; this can be expressed by $g = h^{(p-1)/q} \bmod p$. The user's private key x is randomly generated to be any number from 1 to p-1. The public key y is computed as $y = g^x \bmod p$. Generating the signature of a message M consists of calculating the SHA-1 message hash H(M) and generating a one-time random number k then computing two quantities r and s defined by (See Figure 4):

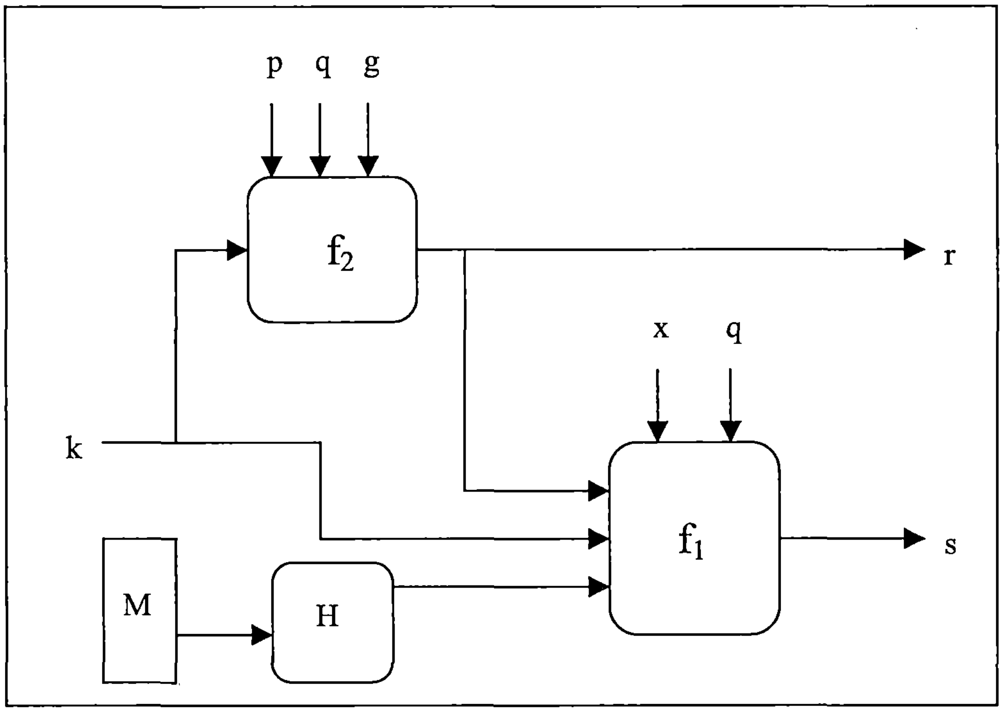


Figure 4. DSA Digital Signature Process

$$s = f_1(H(M),k,x,r,q) = (k^{-1}(H(M)+xr)) \bmod q$$

$$r = f_2(k,p,q,g) = (g^k \bmod p) \bmod q.$$

At the destination, the signature is verified computing a quantity v and comparing it with r; v is defined by $v = ((g^{u1}\ y^{u2}) \bmod p) \bmod q$ where $u_1 = (H(M)w) \bmod q$, $u_2 = rw \bmod q$ and $w = s^{-1} \bmod q$ (See Figure 5).

Pretty Good Privacy

PGP is a hybrid cryptosystem that combines some of the best features of both conventional and public key encryption. Conventional encryption provides fast and

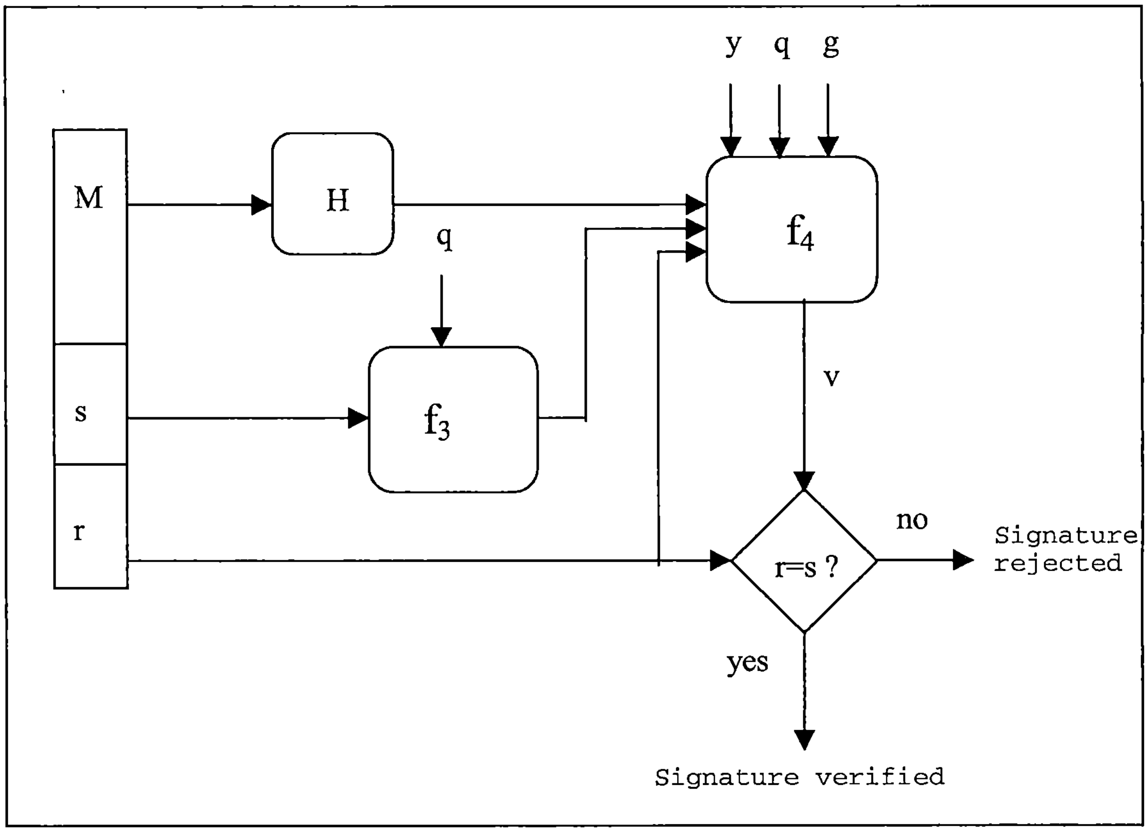


Figure 5. DSA Digital Signature Verification

secure encryption/decryption and public key encryption improves the process of key distribution.

PGP enables users to securely exchange messages, digitally signed documents or secure files in local hard-drives. Confidentiality is the basic service provided by PGP; it allows encrypted messages to be transmitted to a remote recipient or stored in local files. The encryption is performed in the following sequence [8]:

1. The sender generates a random number from mouse movements or keystrokes. This random number is used as a one-time session key that will be used for the current message only.
2. The plaintext is compressed using the ZIP algorithm. The ZIP algorithm is the most commonly used compression algorithm; it makes transferring and copying files faster. The compression saves transmission time and reduces patterns in the plaintext that could be used by a cryptanalysisist to decrypt the message. Decreasing the redundancy in the message makes cryptanalysis of it more difficult. Most cryptanalysis attacks exploit patterns found in the plaintext to break the ciphertext; compression reduces these patterns and thus increases resistance to cryptanalysis. Messages that are too short to compress are not compressed.
3. The plaintext is encrypted with the session key using a secure and fast conventional encryption algorithm, each of the following algorithms can be used: CAST-128, IDEA or Triple DES. At this stage, the ciphertext is produced.

4. The session key is encrypted using RSA with the recipient's public key and transmitted prepended to the ciphertext.
5. At the destination, the receiver recovers the session key using his or her private key.
6. The session key is used to decrypt the ciphertext.

PGP also allows users to exchange authenticated messages; this is done using a digital signature scheme. The following steps describe the process:

1. Using the SHA-1 algorithm, a 160-bit hash code of the message is generated.
2. The hash is encrypted using RSA with the sender's public key then prepended to the message.
3. The receiver recovers the hash code using RSA and the sender's public key.
4. The receiver generates the hash code of the message and compares it with the decrypted hash. The message is authentic if the two hash codes match.

Sometimes, both authentication and confidentiality are required. In this case, a signature for the plaintext is generated and prepended to the message. The message and the signature are then encrypted using the procedure described above.

This combination of two encryption techniques makes PGP perform faster than public key encryption since conventional encryption is about 1,000 times faster than public key encryption [2]. The use of public key encryption to transfer the session key provides a better way of distributing keys between correspondents.

Implementation

The implementation of the security client starts by defining the categories of functions to be provided. Java classes are grouped by type of security function they perform. The next step is to implement the security functions of the security client. Finally, test classes are written to validate the implementation. The security client classes are grouped in the following categories:

- Message digests: this group of classes provides methods to compute a message digest of data.
- Key Pairs: classes under this category provide various functionalities related to key pairs used in public key encryption. Users are provided with methods to generate, import and export keys.
- Encryption and Decryption: this group of classes provides methods to encrypt and decrypt data.

- Digital signature: this group of classes provides methods to sign data and verify digital signatures.
- PGP: classes in this group are used to interface with the key manager service; they are stub classes automatically generated from the web service interface.

CHAPTER FIVE

KEY MANAGER SERVICE DESIGN AND IMPLEMENTATION

Java Cryptography Architecture

The Java language defines a security architecture known as the Java Cryptography Architecture (JCA). The JCA defines APIs that allow developers to incorporate security functionality in their programs. The Java development kit includes APIs for digital signatures and message digests as defined by the JCA. The Java Cryptography Extension (JCE) extends theses APIs to include other security functions for distribution in the United States and Canada only. JCA defines a provider architecture that allows third party implementations to be used. Cryptographic services such as generating signatures or creating message digests are referred to as engines; engines are defined to separate cryptographic services from each other; this way, a provider can choose to only implement a subset of engines defined by the JCA. The engines defined are [6]:

- MessageDigest: used to compute the message digest.

- Signature: used to sign and verify digital signatures.
- KeyPairGenerator: used to generate a pair of public and private key suitable for a specific algorithm.
- KeyFactory: used to translate a key to a key specification and vice versa.
- CertificateFactory: used to create public key certificates.
- KeyStores: used to create and manage databases of keys.
- AlgorithmParameters: used to manage the parameters for a particular algorithm.
- AlgorithmParameterGenrator: used to generate a set of parameters suitable for a specific algorithm.
- SecureRandom: used to generate random or pseudo-random numbers.

New Providers are installed by placing the implementation class files in the class path then adding the reference to the new provider in the java.security file. This is accomplished by adding a new parameter security.provider.n set to the master class name supplied by the provider; it is the provider's master class that always extends the Provider class. The value of n is set to the priority given to the provider. A provider can also be registered programmatically using methods of the Provider class. When an algorithm is requested without specifying the provider's name, the JVM searches the registered providers for an implementation of the algorithm in the same order as the preference set to the providers. In this project, an implementation of the JCE developed by Cryptix will be used. Cryptix is an international organization dedicated to the development of open-source cryptographic libraries; the products development is currently focused on Java [17].

Public Key Management in PGP

Public key management, including the protection of public keys from tampering is the single most difficult problem in applications using public keys [2]. The following scenario illustrates the problem: User A imports a public key associated to User B that was tampered with, and replaced by User C's public key. In this case, User C can forge B's signature in messages sent to A that he or she will accept, and messages encrypted by A and sent to B can be read by C. There is no specific key management scheme provided by PGP, but many options are suggested such as physically delivering public keys to correspondents. If one party knows the other party's public key then he or she can use that public key to send his or her public key encrypted to the other party. Another suggested solution consists of using a mutually trusted entity known as a Certificate Authority (CA), to exchange public keys; the trusted entity would sign the certificates containing the public keys to be exchanged; when a certificate is received, the public key it contains can be considered legitimate since it is signed by the CA. To avoid the need for a CA, PGP introduces the notion of trust. To each public key, a trust level can be associated, this level determines the degree to which the public key owner is trusted to certify or introduce new public keys. Based on all the trust levels known to the server, the legitimacy for each public key can be determined; PGP does not specify how a key legitimacy is computed but suggestions are provided. In the common usage of PGP, a user has a public key ring stored in a local computer. In this project however, all public keys of all users are stored in the server and the end user does not need to keep any information in his or her computer; in fact, users can utilize any computer hosting a client using this API. The key legitimacy can be computed using all users' information stored in the server. Hence, a new public key legitimacy computation algorithm is introduced in this project.

Graph Theory/Dijkstra

A graph consists of a set of points or vertices linked together by a set of edges; a simple example of a graph would be of a computer network where the vertices are computers connected with data transportation media. Formally, a graph is a pair of sets (V, E) where V is the set of vertices and E is the set of edges; a weighted graph is a graph with a cost function C where for each edge (a,b) in E, there is a real number c such that C(a,b) = c. A graph is said to be connected if there exists a path between any two vertices in V. In this project, computing the legitimacy of a public key boils down to solving a shortest path problem. Given a connected and weighted graph, the problem is solved by finding the cheapest path from a source node to a destination node. The algorithm we use is Dijkstra's algorithm; it is an example of a greedy algorithm, at each step, it makes a choice that is best at that moment [13]. The algorithm keeps for each node the cost of reaching it and whether this cost is final. At first, all nodes except to origin are associated a best-estimated cost with the value of 0 and the origin is associated a final cost with a value of 0. At each iteration, the algorithm selects the cheapest node to reach from any node with a final cost, the selected node cost is then set to final; this is repeated until all costs are final. As we can see, Dijkstra's algorithm finds at the same time the shortest path to all nodes in the graph.

Web Services

In general terms, web services are services offered by one application to other applications via the Internet. Web services are usually involved in business-to-business transactions; for example, a company might provide a web site where users can input two addresses and get driving directions from the origin to the destination; the request would be forwarded from the company's server to a remote web service to be processed, then the directions displayed to the user on that company's site. A web service is located based on a URL that is used to send requests. Requests are sent in Extensible Markup Language (XML), XML is a standard that defines a system independent way of representing data. XML is an important part of web services; it makes it possible to interconnect software components written in different programming languages and running on different platforms. Figure 6 describes the relationships between web services components.

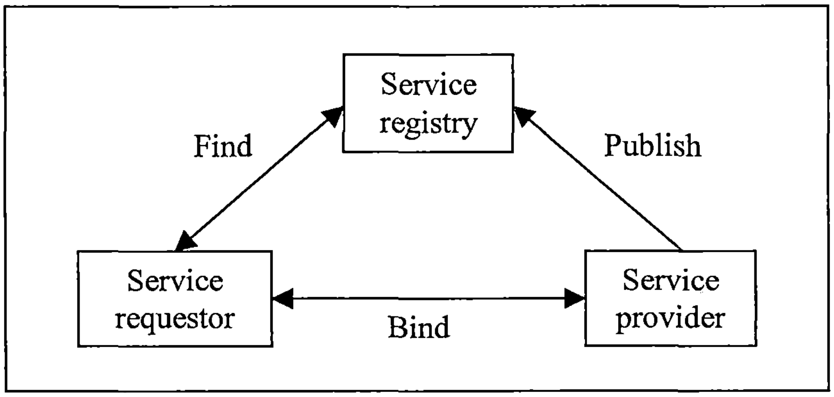


Figure 6. Web Services Components

The process starts by a service provider publishing a service to a service registry, the service is then found in the registry by a service consumer, and the consumer finally binds the service to send requests. Web services are based on the following technologies [15]:

- Web Services Discovery Language (WSDL): it defines a standard way of describing a web service using XML. It contains information about the service such as a description of the methods provided, the abstract description of the data types and URLs that can be used to call the service.
- Simple Object Access Protocol (SOAP): it is a protocol that defines the structure of the XML documents used to send requests and receive responses from a web service. A SOAP document contains a top element envelope; this element contains a header element for properties of the message followed by a body element for the content of the request or the response.
- Universal Description Discovery and Integration (UDDI): it is a specification designed to allow businesses to find each other's services. It defines a way for a service provider to publish a service to a service registry and for service consumers to search a service registry to find services.

E-R Model and Database Design

Based on the requirements specified for the key manager service, the following entities are defined:

- User: this entity defines a user having published a key on the server. The attribute details are shown in Table 2.

Table 2. User Entity

Attribute	Definition	Type
Id	Identifier assigned to the user	Atomic
Name	Name of the user	Composite
E-mail	User's E-mail address (user@xxx.xxx.xxx)	Atomic
Address	User's street address	Composite
Phone number	User phone number	Atomic

- Public key: this entity defines a public key published on the server. The attribute details are shown in Table 3.

Table 3. Public Key Entity

Attribute	Definition	Type
Key	The encoded string representing the public key	Atomic
Key Id	The PGP identifier for the key	Atomic

- Private key: this entity defines a private key published on the server. The attribute details are shown in Table 4.

Table 4. Private Key Entity

Attribute	Definition	Type
Key	The encoded string representing the private key	Atomic
Key Id	The private key identifier	Atomic

The following relationships between entities are identified:

- Trust: this relationship represents a trust from one user to another. It relates a user entity to another. A user may trust and may be trusted by many users. The attribute details are shown in Table 5.

Table 5. Trust Relationship

Attribute	Definition	Type
Truster	Identifier of the trusting user	Atomic
Trustee	Identifier of the user given the trust	Atomic
Level	The level of the trust given to the trustee. Number from 0 to 100	Atomic

- Public key ownership: this relationship binds a user to a public key. It relates a user entity to a public key entity. A user may own no more than one public key and a public key is owned by one user. The attribute details are shown in Table 6.

Table 6. Public Key Ownership

Attribute	Definition	Type
Public Key	Public key Identifier	Atomic
User	Identifier of the key owner	Atomic

- Public key signature: this relationship binds a user entity to a public key entity. A user may sign many public keys and a public key may be signed many users. A user need not sign a public key and a public key need not be signed by a user. The attribute details are shown in Table 7.

Table 7. Public Key Signature

Attribute	Definition	Type
Public Key	Public key Identifier	Atomic
User	Identifier of the key signer	Atomic

- Public key publication: this relationship binds a public key to its publisher. A user may not publish any key or may publish many public keys and a public key is published by one user. The attribute details are shown in Table 8.

Table 8. Public Key Publication

Attribute	Definition	Type
Public Key	Public key Identifier	Atomic
User	Identifier of the user that published the public key	Atomic

- Key pairs: this relationship binds a public key to a private key. It relates a public key entity to a private key entity. A public key may correspond to at most one private key and a private key corresponds to one public key. The attribute details are shown in Table 9.

Table 9. Key Pair Relationship

Attribute	Definition	Type
Public Key	User's public key	Atomic
Private Key	User's private key	Atomic

Figure 7 represents the E-R diagram for the key manager service.

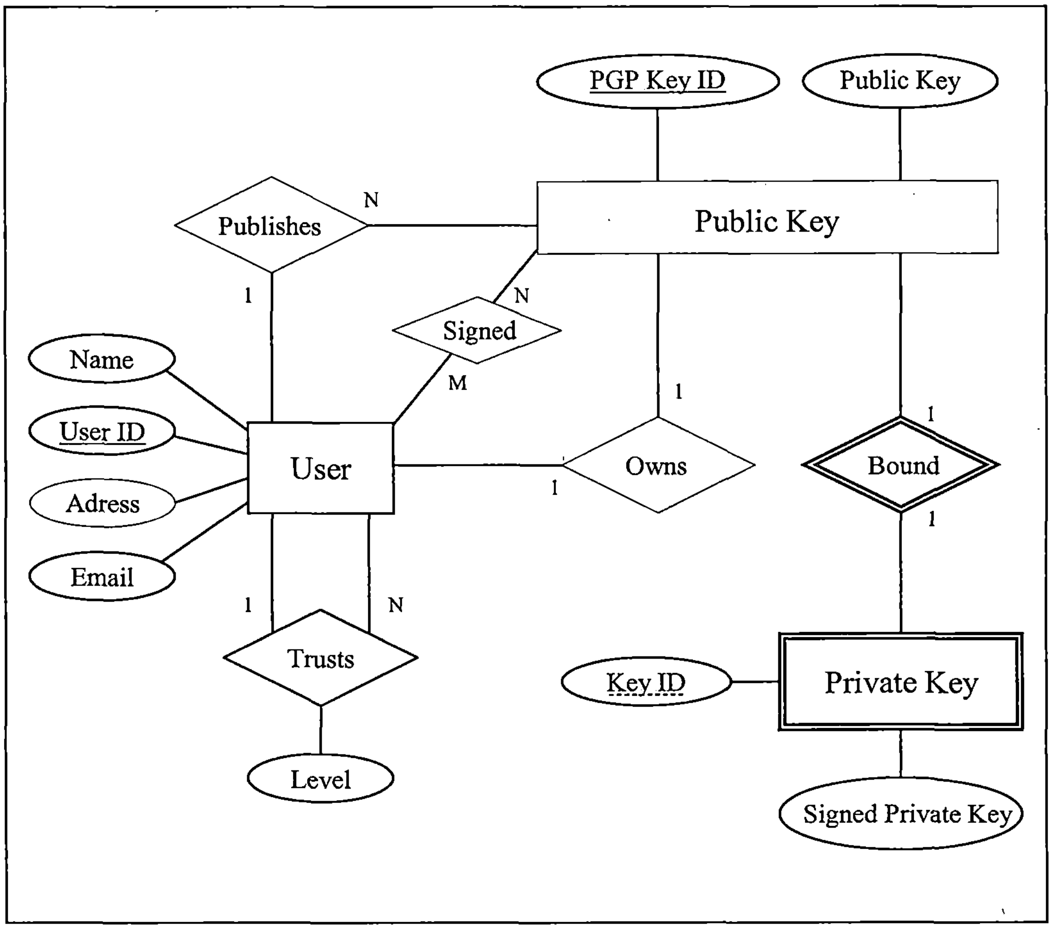


Figure 7. E-R Diagram of the Key Manager Service

Java Database Connectivity

Java Database Connectivity (JDBC) is a programming interface that allows interaction between a Java program and a database. JDBC allows Java programs to connect to relational databases, execute structured query language (SQL) queries and retrieve query results. With JDBC, one can write a database application completely in Java code as opposed to using embedded SQL code that needs to be precompiled before it can be converted into a host-native language like C. JDBC also presents major advantages over Open Database Connectivity (ODBC) developed by Microsoft; utilizing ODBC from Java is possible using what is referred to as a JDBC-ODBC bridge, but this has many drawbacks [16]:

- ODBC is not appropriate for use from Java since it uses a C interface. Calls from Java to native C code affect security, robustness and portability.
- ODBC is harder to learn than JDBC, complex options are used even for simple tasks; JDBC, on the other hand, keeps things simple while allowing complex capabilities to be used when required.
- When using ODBC, the ODBC driver manager and drivers must be manually installed in every machine, whereas JDBC drivers are installed along with the clients on all types of client machines.

Implementation

The implementation of the key manager service includes the following steps:

1. Define the web service API.

2. Write and test a sample implementation of the interface.
3. Deploy the sample service.

Define the Web Service Interface

The key manager service is developed as a web service that fulfills client requests via SOAP. The secure server API is defined by the following service capabilities:

1. Managing users: a client application can create, edit or update users' information on the server.
2. Managing key pairs: the service provides the means to store, update or delete key pairs from the server.
3. Managing public keys: a client application may allow users to store, update or delete public keys belonging to their correspondents.
4. Setting trust levels: this method is used to set a trust level from one user to another, the trust levels are used to compute the level of legitimacy of public keys.
5. Retrieving secret keys for decryption or digital signature.

6. Retrieving public keys for encryption or digital signature verification. A user can also retrieve a public key to distribute to correspondents.
7. Getting a level of legitimacy for a public key.

Write and Test the Interface Implementation

The service implementation of the key manager service interface that we provide manages the users' information and keys using a database as defined previously in the E-R diagram. A public key legitimacy is computed by combining all trust information available on a key K. The first step is to build what we refer to as a trust graph. The first nodes consist of users having signed K; these nodes are all linked to the destination node with a cost of 1. More nodes are added based on trusts assigned to users included in the graph. If the graph contains a node for a user A and B trusts A with a level L then a new node is added for user B linked to A with a weight of L. This process is repeated until no more new nodes can be added. When the trust graph is complete, Dijkstra's algorithm is used to find the longest path to the final destination. The cost of the path to reach the key K from the user's node goes through user B, user G then user I with a cost of 0.729 (=0.9x0.9x0.9x1); in other words, from the user X's point of view, the key K is 72.9% legitimate.

Deploy the Sample Service

In this project, we use Apache's web services framework named AXIS. AXIS is installed first on the Sun One application server based on AXIS' installation guide. Then the web service is deployed. To deploy a web service, a deployment descriptor is required. The deployment descriptor contains instructions to AXIS on how the service should be deployed. In our deployment descriptor, we specify the following parameters: service name, service class, allowed methods and the list of complex types or classes used. The content of the deployment descriptor is listed in Appendix E.

CHAPTER SIX

SAMPLE APPLICATION IMPLEMENTATION

Java Applets

Java applets are programs written in the Java programming language that can be embedded in an HTML page, the same way other components such as images or tables are included. Running applets requires the use of either a Java-enabled browser or a browser with the Java plug-in installed. The Java plug-in enables browsers to run applets using Sun's Java Runtime Environment (JRE) instead of the browser's default. To view a page that contains an applet, the applet's byte code is downloaded from the server to the local system and executed by the browser's JRE or the Java plug-in.

One of the benefits of using Java is the ability to run mobile code. In Java, code is loaded either from the disk or from a remote file system by a Class loader. Class loaders determine how and when classes are added to the running environment making sure of the authenticity of byte code [3]. Every Java VM starts by loading classes from the user's class path using the Primordial class loader; these classes are trusted and not subjected to any verification. Classes can be loaded by other Class Loader Objects such as applet Class Loaders. Applet Class Loaders load classes into a browser by first attempting to load a class using the primordial class loader, if the class is not found then its byte code is downloaded from the remote server via HTTP and examined to ensure that it does not break any security rule but still runs under strict restrictions. The Class loader used in the Java plug-in is referred to as the Plug-in Class Loader. The Plug-in Class Loader allows browsers to accept signed applets to be given the same privileges as local code. When a Plug-in Class Loader detects a signed applet, it prompts the user for permission to run it; the user also has the ability to verify the certificates of the signers. To each Java applet is associated a Code source that consists of the URL from which it was loaded and the list of certificates used to sign it if any. Each class belongs to one and only one protection domain based on its code source [3]; every protection domain has a set of permissions granted to it. An applet can be granted privileges if the user explicitly states additional privileges in a file named .java.policy located in the user's home directory.

For this project, the E-mail client application runs as an applet that is signed to provide these otherwise unauthorized actions (See exhaustive list in appendix A):

- Writing files to the client file system for logging purposes.
- Creating a network connection to a key manager service that can be in a computer other than the host from which the applet originated.
- Using the Cryptix JCE provider instead of the JCE package that is already a part of the client system.

In this project, we assume that the client is using the Java plug-in version 1.4.2 or above.

Java Servlets

Servlets are programs used to build Web pages on a Web server. They are used when the content of the pages to be returned to clients may differ from one request to another. They can be thought of as applets that run on the server side. Building Web pages dynamically for incoming requests is useful in the following cases [14]:

- The request for the Web page depends on data submitted by the user, an example would be a request to a search engine where the user supplies the search keywords.

- The data used to generate a web page change frequently. For example, a traffic report listing the latest traffic incidents using data downloaded periodically from remote sites. If the generated pages content go out of date then new pages are generated.
- The web page requires information from databases or other data sources. For example, a web page that accesses a company's stock needs to be re-generated for each request.

Servlets run on containers also referred to as servlet engines; servlet engines are web server extensions that provide an environment for running servlets [14].

Developing a servlet consists of implementing the servlet interface or extending a class that implements the servlet interface. The servlet interface defines a service method that is called to handle client requests. One such class that implements the servlet interface is HTTPServlet. It provides support for HTTP-specific functionalities such as reading HTTP headers. HTTPServlet is commonly extended to implement servlets. HTTPServlet is an abstract class with additional methods called by its service method; these methods must be implemented by classes extending HTTPServlet:

- doGet: to handle HTTP GET requests.
- doPost: to handle HTTP POST requests.
- doPut: to handle HTTP PUT requests.
- doDelete: to handle HTTP DELETE requests.
- doHead: to handle HTTP HEAD requests.
- doOptions: to handle HTTP OPTIONS requests.
- doTrace: to handle HTTP TRACE requests.

Servlets are Java's substitution for CGI programming. They offer many advantages over traditional CGI scripts. Java servlets are more portable since they take advantage of Java's "Write Once, Run Anywhere" feature. They can also run on any web server or servlet Container thanks to a well-defined servlet API. They are more efficient than CGI scripts. As opposed to CGI technology where a new process is started for every request, Java servlets fulfill each request by spawning one lightweight Java thread per request; this eliminates overhead of starting a new operating system process. All servlet requests run under the same Java VM making it possible to cache information in memory for future use; CGI scripts must use a database or a file to accomplish a stateful mode of operation. Java servlets can also communicate with the web server, something CGI scripts cannot do; a servlet can write to a web server log file, share resources such as message queues and database connection pools with other servlets.

Java Server Pages Technology

Java Server Pages (JSP) are HTML pages that contain Java code to generate dynamic content. Before providing a JSP page to clients, the JSP container first translates the JSP page into a servlet class that will be used to produce the same output. Java code within a JSP page can be thought of as part of the implementation of the servlet's doGet or doPost method; objects such as requests, responses, sessions and servlet contexts are available within the JSP page. A JSP page does not do anything a servlet cannot do but it makes it more convenient to write the HTML code without having to use Java statements; HTML code is written as it would be done in a usual HTML page. The major advantage of using JSP is to separate the development and the authoring roles; developers write the components that perform the processing to generate the content and authors write the HTML code for the presentation.

JavaMail API

The JavaMail API is designed to provide a protocol-independent package for reading and sending electronic messages [12]. JavaMail does not perform the actual transporting, delivering or forwarding of messages; it rather relies on mail servers to perform the actual transfer of messages. JavaMail communicates with mail servers using providers; it comes with providers for SMTP, POP3 (Post Office Protocol 3) and IMAP mail protocols. Adding support for a mail protocol requires implementing the provider interface specifically for the protocol. In our project, we will limit access to POP3 mail servers. It is a widely used protocol for retrieving E-mails. It defines the communication protocol between a POP3 E-mail client and a POP3 mail server. The POP3 client/server communication consists of three phases [12]: authentication, transaction and update. The client starts by sending the username and password to authenticate the owner of the mailbox; if the authentication succeeds then the user proceeds to the transaction phase. During the transaction phase, the user's mailbox is locked by the server; only a single client can connect to a mailbox at a time. The user holding the connection to a mailbox sends POP3 commands to the server, ending with an update command that closes the connection. After closing the connection, the server updates the user's mailbox to reflect the changes he or she requested during the transaction phase.

Server Implementation

The server is implemented as a servlet that processes requests originating from either the browser or the applet. For example, when the user logs in, the browser sends a request to the servlet that returns the response by forwarding to a JSP page; the applet can make a request to the servlet to retrieve a public key needed for encryption. When a request arrives, the servlet starts by reading a parameter named 'cmd' which specifies the action that needs to be taken. Each action is implemented as a class that implements the IAction interface. The servlet fulfills requests by getting an IAction instance from the ActionFactory using the cmd string then calling the process method on the IAction instance (See Figure 9).

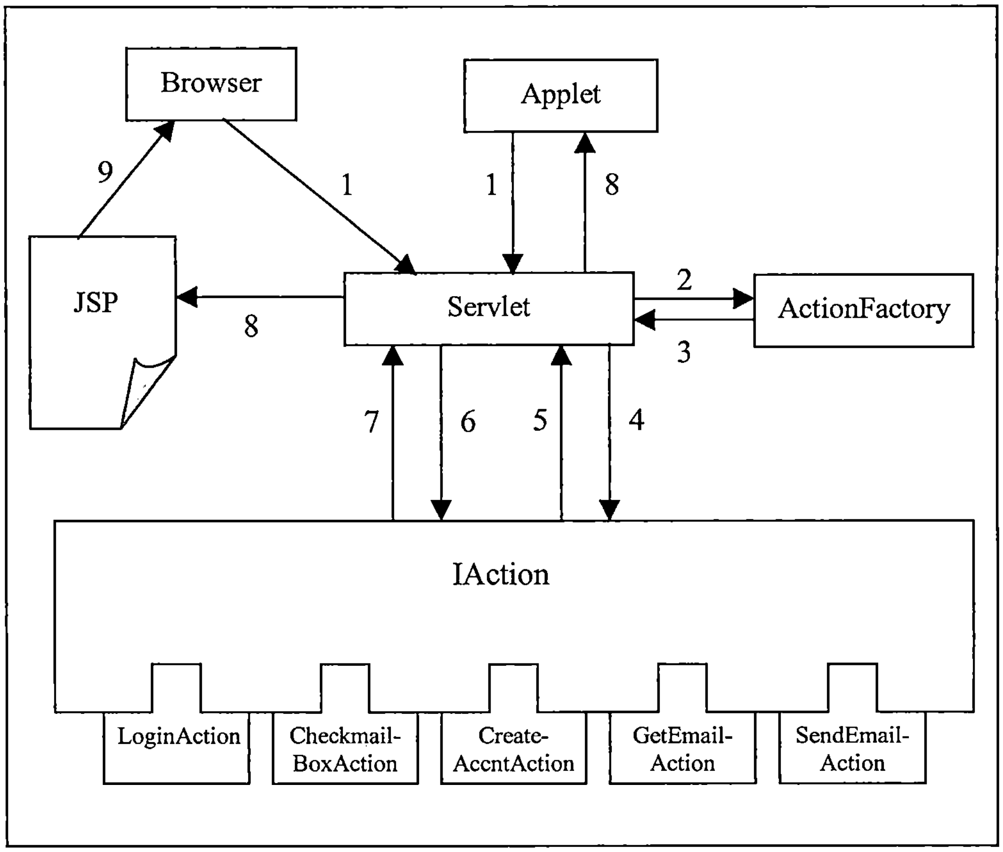


Figure 9. Structure of the E-mail Servlet

As shown in Figure 9, the following steps are taken to fulfill incoming requests:

1- Browser or applet sends a request to the servlet.

2- The servlet sends a getInstance request to the ActionFactory by passing in the cmd parameter.

3- The ActionFactory returns an instance I of IAction interface.

4- The servlet sends a process command to I.

5- I processes the command and returns control to the servlet.

6- The servlet asks I for the response type.

7- I returns the response type to the servlet.

8- If the response type is T, the servlet forwards to the JSP named T. If T is null, the servlet returns the content directly to the applet.

The Action factory getInstance method can be visualized using a Nassi-Shneiderman [18] diagram in Figure 10:

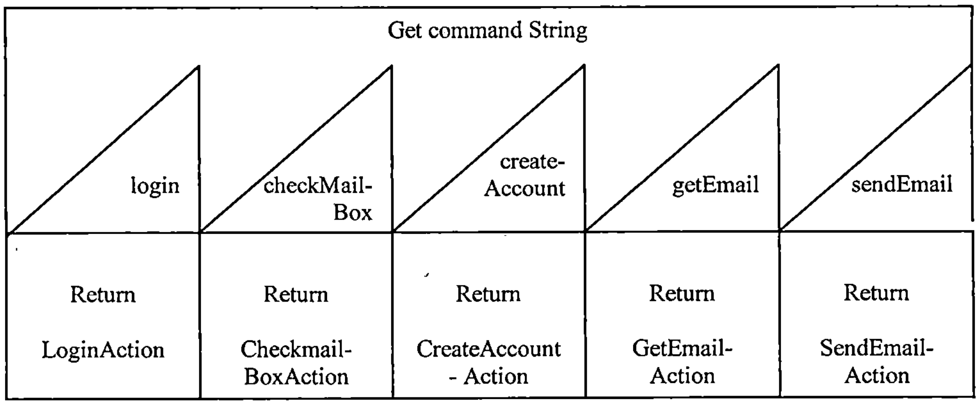


Figure 10. Action Factory GetInstance Method

Client Implementation

The E-mail client consists of a Java applet that utilizes the security client. The Graphic User Interface (GUI) is built using Java Swing components. The GUI contains four tabs for checking a mailbox, sending E-mails, publishing keys and managing address books.

Mailbox Tab

The mailbox-tab is used to view the content of the user's mailbox. When the user selects this tab, the applet sends an HTTP request to the servlet to retrieve the content of the mailbox. The servlet replies with a list of comma-separated encoded Strings; these strings contain the sender, the title, date received and an ID for each E-mail. The applet parses this response and displays the E-mail list on the screen. When the user clicks on the Open E-mail button, the applet checks the user's private key. If it was not yet retrieved then a request is sent to the key manager service for the user's private key. The user is then prompted for the passphrase to recover the private key, which is cached by the applet for later use. The applet then sends an HTTP request to the servlet containing the E-mail ID; the servlet fetches the E-mail body from the database and writes it back to the applet. The applet checks whether the E-mail body is encrypted. If needed, the message is decrypted and displayed to the user in a popup window.

Publish a New Key Tab

This publish-new-key-tab allows a user to publish a new public key to the server. The user starts by pasting the public key in the text area. The applet then inspects the key for any certificates and displays the issuers on the screen. The user has the option to assign a trust level to each certificate issuer as well sign the new key. When the user submits the key, the applet sends requests to the security manager service to publish the new key and save the new trust information if any.

Compose E-mail Tab

The compose-E-mail-tab allows a user to compose or reply to an E-mail. When the user inputs the recipient's E-mail address, the applet sends a request to the key manager service for the recipient's key legitimacy. If the recipient is unknown then the user is asked to submit the key; otherwise, the recipient's key legitimacy is displayed on the screen. When the user clicks the send button, the applet encrypts the E-mail body using the recipient's public key then sends an HTTP request to the servlet to deliver the E-mail. The servlet opens a session with the user's outgoing mail server to sends the message.

Address Book Tab

This Address-book-tab provides users with an interface to manage address books. It allows users to add new correspondents, delete correspondents or update information about their correspondents. These actions are accomplished by the applet sending an HTTP request to the servlet, which makes the necessary updates to the database.

Creating a New Account

To create a new account, a separate applet is loaded. The user fills in information required to create the new account such as username and password. When the user clicks the create new account link, the applet opens a small window and the user is asked to keep moving the mouse until it closes. The random movements of the mouse are used to generate a seed for a random number generator, which is needed to generate the key pair for the new user. The applet then sends the information in the form to the servlet through an HTTP request. The servlet saves the information about the new user account such as the user name, SHA1 digest of the password, PGP passphrase and incoming mail server to use. The servlet also sends the key pair to the web service to be stored on the server. The passphrase used to encrypt the private key is not sent to the servlet or to the service.

To each one of the tabs corresponds a separate Java class for its implementation. The separation of functionalities simplified implementation and debugging.

CHAPTER SEVEN

TESTING THE API

JUnit

JUnit is a simple framework to write unit tests for Java programs. While testing using a debugger to evaluate expressions or printing out debug messages requires the programmer's analysis and interpretation, JUnit tests are easy to run and do not require the programmer to analyze any information. Using JUnit, debugging can be done without stepping through the code with a debugger in order to reach a statement or adding statements in the code to print out debug messages. To write a test using JUnit, the TestCase class is extended and its runTest method is overridden. At any stage of the test, checks can be added to verify that an expression matches its expected value by calling the assertTrue method. In this project, all unit tests are performed by JUNIT.

Unit Tests

The API cannot be considered for practical use unless it is tested and validated. The following tests were successfully performed:

- Generated symmetric encryption key.
- Generated public key pair.
- Encrypted and decrypted data using public and private keys.
- Encrypted and decrypted data using a secret key.
- Digitally signed data.
- Verified digital signature.
- Generated message digests.

Table 10 summarizes the unit test results.

Integrity Tests

During the development phase, the system is broken down into smaller and simpler units that can be implemented separately. Although the tests on individual units were ran successfully, there is no guarantee that the system will perform as desired once the components are put together. The integrity tests verify the functionality of the system as a whole. Our integrity tests validate the interfacing between the security client and the key manager service as well as integrity with other PGP software. PGP 8.0 for Microsoft Windows

Table 10. Unit Test Results

	Tests Performed	Results
IDEA	• Generate IDEA Key • Encrypt data • Decrypt Data	*Pass*
DES	• Generate DES Key • Encrypt data • Decrypt Data	*Pass*
TripleDES	• Generate TripleDES Key • Encrypt data • Decrypt Data	*Pass*
RSA Key Pairs	• Generate Key Pair • Encrypt data • Decrypt Data • Export key Pair to files • Import key Pair from files	*Pass*
ElGamal Key Pairs	• Generate Key Pair • Encrypt data • Decrypt Data • Export key Pair to files • Import key Pair from files	*Pass*
DSA Digital Signature	• Digitally sign data • Verify Digital signature	*Pass*
MD5WithRSA Digital Signature	• Digitally sign data • Verify Digital signature	*Pass*
SHA1withRSA Digital Signature	• Digitally sign String • Verify Digital signature	*Pass*
SHA, SHA0 and SHA1 Message Digests	• Generate Message digests • Validate Message digests	*Pass*
MD2, MD4 and MD5 Message Digests	• Generate Message digests • Validate Message digests	*Pass*

trial version is used for the tests. The following integrity tests were performed:

- Made calls from the client to the server and verify the results.
- Generated key pairs and imported them using PGP.
- Encrypted text using the security client and decrypted using PGP.
- Encrypted text with PGP and decrypted it using the security client.
- Encrypted text using a given username used to import the public key from the server.
- Decrypted text using a given username used to import the private key from the server.
- Signed text given a username used to import the user's private key from the server
- Signed a public key given a username used to import the user's private key from the server.

The Table 11 summarizes the test results.

Table 11. Integrity Test Results

	Tests Performed	Results
Key Pairs	• Publish key pair on the server • Encrypt data using the security client then decrypt using a PGP software • Encrypt data using a PGP software then decrypt using the security client	*Pass*
Digital Signature	• Digitally sign data using a key imported from the server • Digitally sign data using security client then verify signature in PGP • Digitally sign data using PGP and verify signature using the security client	*Pass*
Public keys	• Retrieve a pubic key from the server given a user name • Publish a new signed public key on the server • Assign a trust level to a public key owner • Retrieve a level of legitimacy of a pubic key	*Pass*

CHAPTER EIGHT

USER MANUAL

Security Client

The security client provides methods to perform the following security functions: encryption, decryption, digital signature and message digest generation. Classes are grouped by functionality into different packages.

Message Digests

The message digest classes are grouped under the scapi.md package. This package contains one class for each message digest algorithm. The content of the package consists of the following: MD2.java, MD4.java, MD5.java, SHA.java, SHA0.java and SHA1.java. To compute a message digest of string s or a byte array b using an algorithm A, make the following call: A.digest(s) or A.digest(b).

Key Pairs

Two types of key pairs are supported: ElGamal and RSA. ElGamal and RSA key pairs are implemented in scapi.kp.elgamal.ElGamalKeyPair and scapi.kp.rsa.RSAKeyPair respectively. Both classes implement the IkeyPair interface; therefore other implementations of key pairs can be used. From the programmer's prospective, there are only two relevant Java classes: IKeyPair and KeyPairFactory. To keep this level of abstraction and simplicity, specific features of algorithms such as algorithm parameters will not be available. To generate a key pair, an instance of IKeyPair is created using the KeyPairFactory by passing to its getInstance method the algorithm name; currently, valid names are RSA and ElGamal. On the IKeyPair instance, the genKeys method is then called; this method takes three parameters: a string to be used as a random seed that can be null, a username and a passphrase. The IKeyPair interface provides two methods: getPublicKey and getPrivateKey that can be used to access a public or a private from a key pair.

Encryption and Decryption

Encryption and decryption methods are implemented in the scapi.enc package. In the case of conventional encryption, the ConvEnc class is used, whereas the PubEnc class is used for public key encryption. Both classes provide an encrypt method for encryption and a decrypt method for decryption.

Digital Signature

Digital signature related functions can be performed using the scapi.signature.DigitalSignture class. To sign data, the sign method is called. Digital signatures are verified by invoking the verify method.

Key Manager Service

Deploying the Key Manager Service

The key manager service is deployed as a SOAP web service. When publishing the key manager service, the Service interface implementation class is specified in the deployment descriptor. In this project, we use the Apache AXIS framework for web services. The service can be deployed by writing the deployment descriptor then using the AdminClient program that is part of the Apache AXIS. The deployment descriptor and a script we wrote in this project for publishing the service can be used for reference.

Extending the Key Manager Service

The default implementation of the key manager service provided uses a database for persistence and the public key management scheme that we designed. But the key server can be extended for instance to save the public keys in a remote key server. There are two approaches that can be undertaken to extend the service:

- Extend the existing class DefaultSecurityManagerImpl and override only chosen methods. For example, a new implementation can reuse the database we designed but implement a new method to compute the level of legitimacy of a public key; in this case, the getKeyLegitimacy method is overridden.
- Implement a new key server by writing a class that implements ISecurityManager. One can always extend SecurityManagerUtil for useful and reusable methods.

Once, the new class is written and tested, it can be plugged in as the implementation for the web service. This is done by setting the class name parameter in the deployment descriptor to the fully qualified name of the new class.

E-mail Application

Deploying the E-mail Servlet

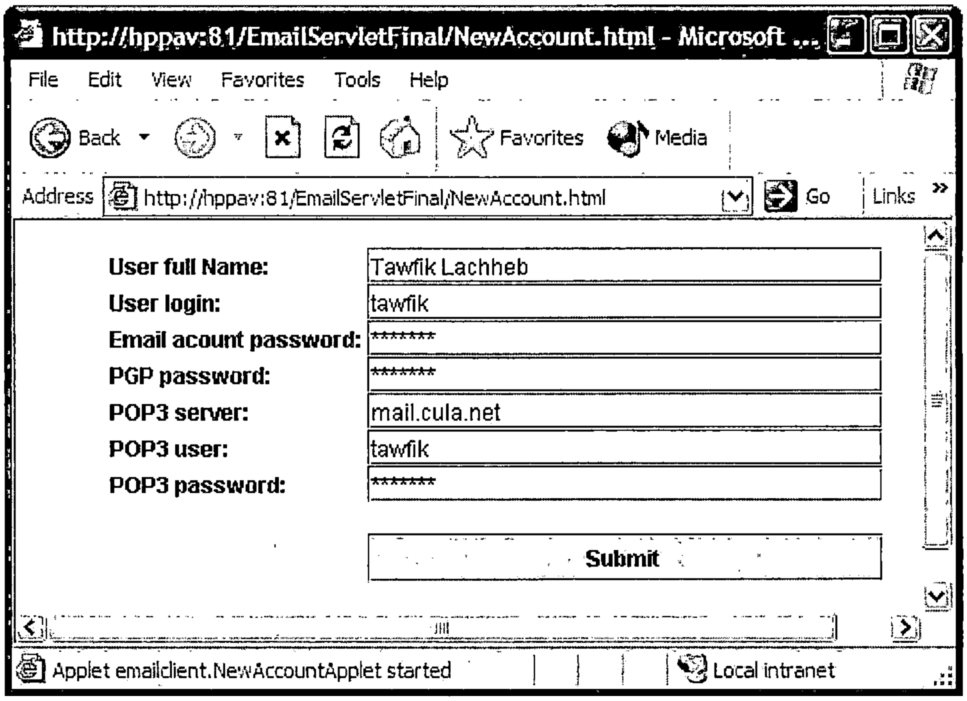
The E-mail servlet can be deployed on any Java 2 Enterprise Edition (j2ee) compliant application server. We tested the deployment on Sun One Application Server Version 7 and Apache Tomcat Version 4.1. The servlet is packaged in the standard web application structure defined by the servlet specification; an ANT script is provided to re-generate the Web Archive (WAR) file in case changes are made in the servlet code. Depending on the application server used, different steps are followed to deploy the web application. In general, the steps include:

- Giving a context for the application. The application context is the path used by clients to remotely access the web application.
- Providing the location of the WAR file used to load the web application on the server.
- Assigning a name for the web application.

Using the E-mail Client

Before using the E-mail client, the user starts at the login page. If the user is new, then he or she needs to create a new account. Once the user logs in, the E-mail client applet is loaded. The E-mail client applet allows a user to manage the mailbox, read and send E-mails, publish public keys to the server and manage the address book. The E-mail client works as described below:

- Creating a new account: to create a new account, the user clicks on the new account link in the login page. The new account applet is then loaded; it contains the following fields shown in Figure 11:


http://hppav:81/EmailServletFinal/NewAccount.html - Microsoft ...
File Edit View Favorites Tools Help
Back
Favorites
Media
Address
http://hppav:81/EmailServletFinal/NewAccount.html
Go
Links
Submit
Applet emailclient.NewAccountApplet started
Local intranet

User full Name:	Tawfik Lachheb
User login:	tawfik
Email acount password:	*******
PGP password:	*******
POP3 server:	mail.cula.net
POP3 user:	tawfik
POP3 password:	*******

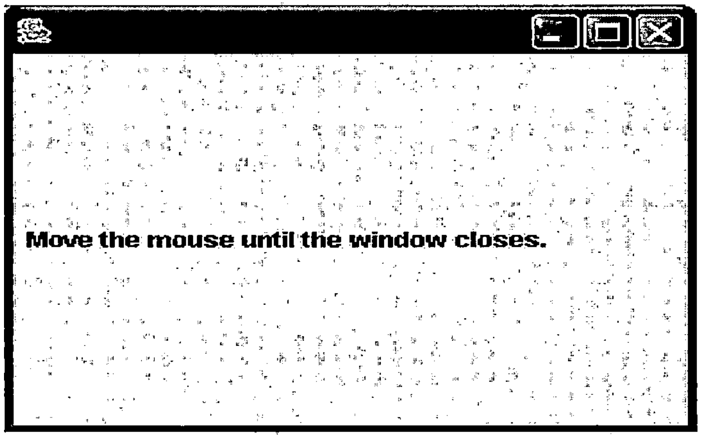
Figure 11. New Account Screen

  - User full name: the user's first, middle and last name as it is desired to appear while using the application. It will also be used along with the E-mail address to identify the owner of a public key. The standard way of naming a public key owner is used. If the user full name is 'Tawfik Lachheb' and the E-mail address is 'tlachheb@csci.csusb.edu' then the owner for the public key is identified by 'Tawfik Lachheb <tlachheb@csci.csusb.edu>'.

- User Login: the login the user chooses for the server.
- User Password: The password used for authentication on the server. Only a message digest of this password will be sent to the server, the server will authenticate a user by comparing the hash of the password submitted from the login page with the password hash stored in the database for the user.
- PGP passphrase: the passphrase that will be used to encrypt the private key, this passphrase will not be sent to the server for security reasons.
- POP3 server: the server used to check for incoming E-mails.
- POP3 user: the user name for the account on the POP3 server.
- POP3 password: the password used to authenticate on the incoming mail server, this

password will be stored by the servlet in plain text.

Once the user fills out the fields and clicks on the submit button, a small window will appear asking the user to keep moving the mouse (See Figure 12).

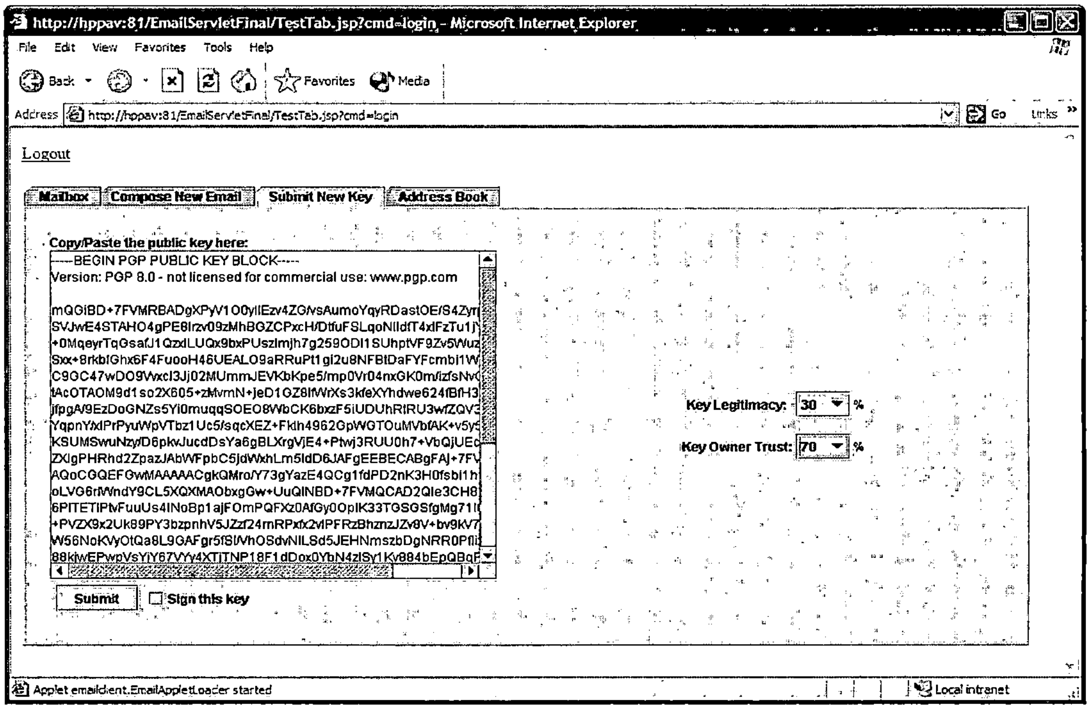


Move the mouse until the window closes.

Figure 12. Random Seed Generation

This is to generate a seed for the random number generator. Once enough random mouse locations on the window are captured, the window will close itself and the key pair will be generated. At this stage, the applet sends the content of the form along with the new key pair to the server.

- Publishing a new key: a new public key can be published in the Submit New Key tab shown in Figure 13. Initially, it contains one large text area


http://hppav:81/EmailServletFinal/TestTab.jsp?cmd=login - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Favorites Media
Address http://hppav:81/EmailServletFinal/TestTab.jsp?cmd=login Go Links
Logout
Mailbox
Compose New Email
Submit New Key
Address Book
Submit
Applet emailclient.EmailAppletLoader started
Local intranet

Copy/Paste the public key here:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PGP 8.0 - not licensed for commercial use: www.pgp.com

mQGiBD+7FVMRBADgXPyV1O0ylIEzv4ZG/vsAumoYqyRDastOE/S4Zyr
SVJwE4STAHO4gPE8lrzv09zMhBGZCPxcH/DtfuFSLqoNIldfT4xFzTu1j
+0MqeyrTqGsafJ1QzxlLUQx9bxPUszImjh7g259ODI1SUhptVF9Zv5Wuz
Sxx+8rkbIGhx6F4FuooH46UEALO9aRRuPt1gl2u8NFBtDaFYFcmbl1W
C9GC47wDO9Wxcl3Jj02MUmmJEVKbKpe5/mp0Vr04nxGK0m/izfsNv
tAcOTAOM9d1so2X605+zMvmN+jeD1GZ8lfWrXs3kfeXYhdwe624fBfH3
jfpgA/9EzDoGNZs5Yi0muqqSOEO8WbCK6bxzF5iUDUhRIRU3wfZQV3
YqpnYxlPrPyuWpVTbz1Uc5/sqcXEZ+Fklh4962GpWGTOuMVbfAK+v5y
KSUMSwuNzy/D6pkvJucdDsYa6gBLXrgVjE4+Ptwj3RUU0h7+VbQjUEc
ZXlgPHRhd2ZpazJAbWFpbC5jdWxhLm5ldD6JAFgEEBECABgFAJ+7FV
AQoCGQEFGwMAAAAACgkQMro/Y73gYazE4QCg1fdPD2nK3H0fsbl1h
oLVG6rWndY9CL5XQXMAObxgGw+UuQINBD+7FVMQCAD2Qle3CH8
6PITETIPtvFuuUs4INoBp1ajFOmPQFXz0AfGy0OpIK33TGSGSfgMg71I
+PVZX9x2Uk89PY3bzpnhV5JZzf24mRPxfx2vlPFRzBhznzJZv8V+bv9KV7
W56NoKVyOtQa8L9GAFgr5fSIVhOSdvNILSd5JEHNmszbDgNRR0Pfli
88kiwEPwpVsYiY67VYy4XTjTNP18F1dDox0YbN4zlSy1Kv884bEpQBqF
```

Key Legitimacy: 30 %

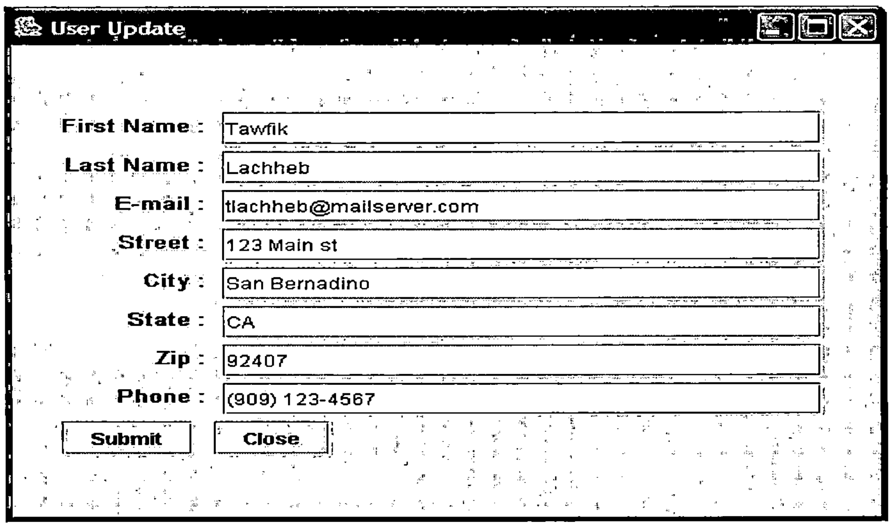
Key Owner Trust: 70 %

[ ] Sign this key

Figure 13. Public Key Submission Screen

where the user pastes the public key received from a correspondent. The user then can assign a level of legitimacy to the new key and a trust level to its owner. The user also has the option to sign the key before submitting it. Once the key is submitted, the user can start sending E-mails to the key owner.

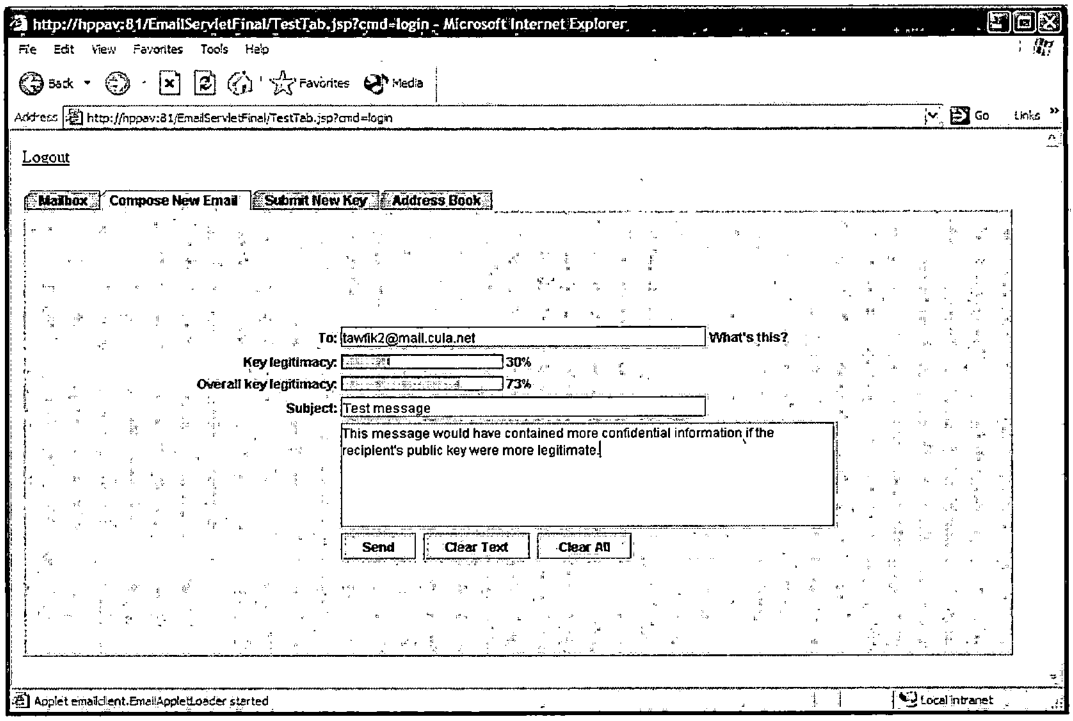
- Managing address books: the content of an address book can be viewed in the Address Book tab. The user has the ability to delete, update or add new entries (See Figure 14).



First Name :	Tawfik
Last Name :	Lachheb
E-mail :	tlachheb@mailserver.com
Street :	123 Main st
City :	San Bernadino
State :	CA
Zip :	92407
Phone :	(909) 123-4567

Figure 14. Contact Update Screen

- Composing E-mail: to compose a new E-mail, the user switches to the Compose New E-mail tab. This tab contains the following fields as shown in Figure 15:
  - To: the recipient's E-mail Address.
  - Key legitimacy: the legitimacy level of the key if assigned by the user.
  - Overall key legitimacy: the legitimacy of the key computed by aggregating the trust information collected on all users as described previously in this document.
  - Subject: the subject for the E-mail.
  - Body: the actual content of the E-mail.


http://hppav:81/EmailServletFinal/TestTab.jsp?cmd=login - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Favorites Media
Address http://hppav:81/EmailServletFinal/TestTab.jsp?cmd=login
Go Links
Logout
Mailbox
Compose New Email
Submit New Key
Address Book
What's this?
Send
Clear Text
Clear All
Applet emailclient.EmailAppletLoader started
Local intranet

To: tawfik2@mail.cula.net

Key legitimacy: 30%

Overall key legitimacy: 73%

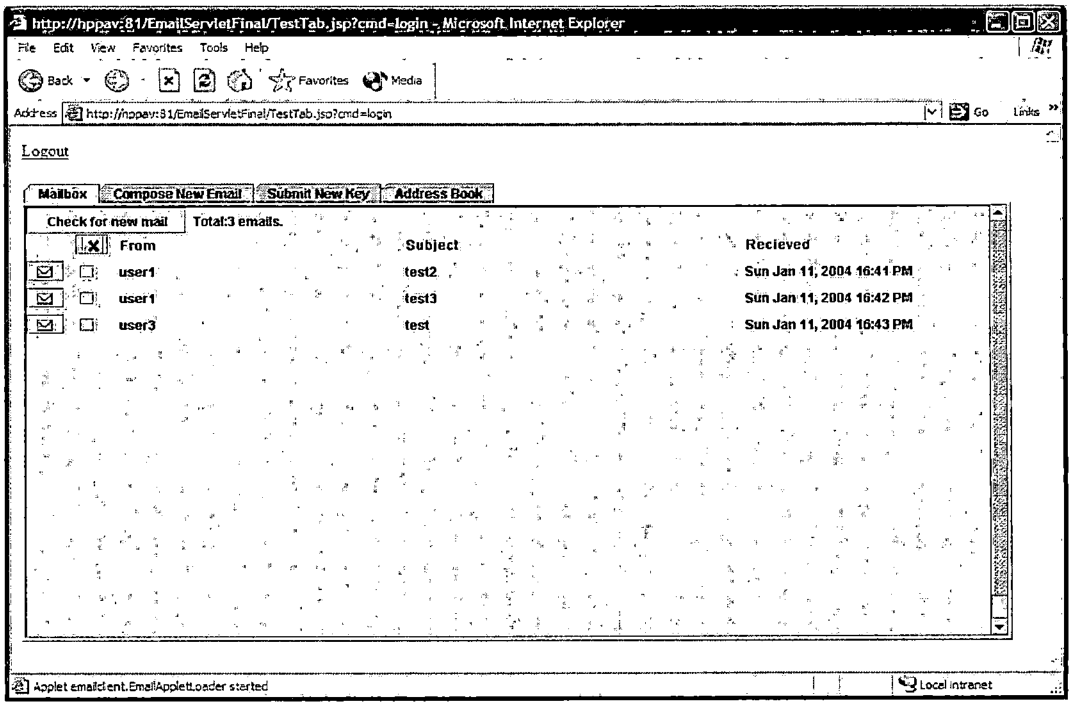
Subject: Test message

This message would have contained more confidential information if the recipient's public key were more legitimate.

Figure 15. Composing E-mail Screen

Once the user inputs or updates the recipient's E-mail address, the applet retrieves the key legitimacy and the overall key legitimacy from the server; these and their numeric values are then displayed on the two scale bars. If the key is not available in the server then the submit button is disabled and the following message appears to the user: 'There is no public key for this E-mail address on the server, please submit it first'. The user writes the subject and the E-mail body after evaluating the legitimacy of the key and deciding whether to send the E-mail or not. Also based on the legitimacy, the user might change the content to include more or less sensitive information.

- Checking a mailbox: the content of the mailbox can be checked when switching to the Mailbox tab shown in Figure 16. The applet submits an HTTP request to the servlet containing the username of the mailbox owner. The servlet connects to the user's incoming mail server to check for new E-mails. New E-mails are retrieved and stored in the database. The servlet replies to the applet with the list containing the E-mails addressed to the user.


http://hppav:81/EmailServletFinal/TestTab.jsp?cmd=login - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Favorites Media
Address http://hppav:81/EmailServletFinal/TestTab.jsp?cmd=login Go Links
Logout
Mailbox
Compose New Email
Submit New Key
Address Book
Check for new mail
Applet emailclient.EmailAppletLoader started
Local intranet

Total:3 emails.

From	Subject	Recieved
user1	test2	Sun Jan 11, 2004 16:41 PM
user1	test3	Sun Jan 11, 2004 16:42 PM
user3	test	Sun Jan 11, 2004 16:43 PM

Figure 16. Mailbox Screen

CHAPTER NINE

CONCLUSION

Summary

Applications can be extended with security features using the client/server API we developed in this project. Applications can provide users with encryption/decryption, digital signature, message digest and key management functionalities. The client side of the API consists of a library of security methods as well as methods to communicate with the server side. The server side of the API is deployed as a SOAP web service that exposes an interface for key management. This API was developed to be very flexible, it allows custom implementations to be plugged-in. The default implementation we provide uses MySQL version 4.0.15 and was tested with PostgresSql version 7.3.4, other database systems can be easily used by simply editing property files. In this project we introduced a method to compute the level of legitimacy of a public key; the process consists of solving a longest path problem from one node to another in a graph. We also developed a secure Internet-based E-mail application that uses the API. It allows users to exchange encrypted emails, publish keys on the server and manager their address books. This sample application can be considered as a reference when using the API.

Looking Forward

The secure client/server API we developed can be extended to secure the communication between the client and the server. Currently, requests are being sent between the client and the server unprotected. For example, an intruder can send a request to the server to update a public key replacing it with his or her key; in this case, the intruder would be reading information confidential to other users.

The client server communication can be secured using the Secure Socket Layer Protocol (SSL) developed by Netscape. SSL is a protocol commonly used by browsers to securely communicate with web servers while transferring sensitive information such as credit card numbers or social security numbers. By deploying the key manager web service with an HTTPS end point, the communication with the client could be secured without having to make any changes in the implementation of the API.

Another approach to secure the client/server communication consists of using the Web Services Security (WSS) protocol. WSS is a flexible extension to the SOAP protocol that adds security features. WSS defines a SOAP header element called security token [18]. A security token can contain a username and password. Adding a security token to requests from the security client is not sufficient in our case because of the threat of replay attacks. The replay attack problem is often solved using what is referred to as a nonce. A nonce is a continuously changing string that originates from the server and is returned by the client with the request; it is used to inject randomness in encrypted or message-digested information. An example of a security token that can be used for a given password P and a nonce N consists of the username and the password containing the encrypted NP using the user's private key.

Even though we have developed a client/server API in this project, applications can make use of the key manager service only. Custom applications written in other programming languages can invoke the key manager service using automatically generated client stubs.

The sample E-mail application was developed with more emphasis on using the security client than implemnting E-mail client functionalities. This is to provide a good example for the use of the API; the sample application can be extended to include more common features in E-mail clients.

APPENDIX A

RESTRICTONS ON APPLETS WITHIN THE JAVA SANDBOX

- Read files on the client file system.
- Write files to the client file system.
- Delete files on the client file system, either by using the File.delete method, or by calling system-level rm or del commands.
- Rename files on the client file system, either by using the File.renameTo method, or by calling system-level mv or rename commands.
- Create a directory on the client file system, either by using the File.mkdirs methods or by calling the system-level mkdir command.
- List the contents of a directory.
- Check to see whether a file exists.
- Obtain information about a file, including size, type, and modification timestamp.
- Create a network connection to any computer other than the host from which it originated.
- Listen for or accept network connections on any port on the client system.
- Create a top-level window without an untrusted window banner.
- Obtain the user's username or home directory name through any means, including trying to read the system properties: user.name, user.home, user.dir, java.home, and java.class.path.
- Define any system properties.
- Run any program on the client system using the Runtime.exec methods.
- Make the Java interpreter exit, using either System.exit or Runtime.exit.
- Load dynamic libraries on the client system using the load or loadLibrary methods of the Runtime or System classes.
- Create or manipulate any thread that is not part of the same ThreadGroup as the applet.
- Create a ClassLoader.
- Create a SecurityManager.
- Specify any network control functions, including ContentHandlerFactory, SocketImplFactory, or URLStreamHandlerFactory.
- Define classes that are part of packages on the client system.

APPENDIX B

SAMPLE CLIENT CODE

```
package scapi.enc;

import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.util.Collection;
import cryptix.message.EncryptedMessageBuilder;
import cryptix.message.LiteralMessage;
import cryptix.message.LiteralMessageBuilder;
import cryptix.message.Message;
import cryptix.message.MessageFactory;
import cryptix.openpgp.PGPArmouredMessage;
import cryptix.pki.KeyBundle;
import cryptix.message.EncryptedMessage;

public class ConvEnc {

  /**
   * This mehod encrypts a message using a symeric key.
   *
   * @param p_msg message to be encrypted
   * @param p_publicKey key to be used for the encryption
   * @return The encrypted message.
   * @throws Exception
   */


  public static String encrypt(String p_msg, KeyBundle
p_publicKey) throws Exception {
    LiteralMessage litmsg = null;
    LiteralMessageBuilder lmb =
LiteralMessageBuilder.getInstance("OpenPGP");
    lmb.init(p_msg);
    litmsg = (LiteralMessage) lmb.build();

    Message msg = null;
    EncryptedMessageBuilder emb =
EncryptedMessageBuilder.getInstance("OpenPGP");
    emb.init(litmsg);
    emb.addRecipient(p_publicKey);
    msg = emb.build();

    PGPArmouredMessage armoured;
    armoured = new PGPArmouredMessage(msg);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    out.write(armoured.getEncoded());
    out.close();
    return new String(out.toByteArray());
  }
  /**
   * This method decrypts a message using a symeric key.
   *
   * @param p_msg encrypted armoured message.
   * @param p_secretKey symetric key used for decryption.
   * @param p_passwd password to recover the key
   * @return decrypted message
   * @throws Exception
   */
  public static String decrypt(String p_msg, KeyBundle
p_secretKey, String p_passwd) throws Exception {

    MessageFactory mf = MessageFactory.getInstance("OpenPGP");
    ByteArrayInputStream in = new
ByteArrayInputStream(p_msg.getBytes());
    Collection msgs = mf.generateMessages(in);
    EncryptedMessage em = (EncryptedMessage)
msgs.iterator().next();
    in.close();

    Message msg = null;
    try {
      msg = em.decrypt(p_secretKey, p_passwd.toCharArray());
    } catch (Exception p_e) {
      p_e.printStackTrace();
      throw p_e;
    }
    return ((LiteralMessage) msg).getTextData();
  }
}

package scapi.enc;

import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.util.Collection;
import cryptix.message.EncryptedMessageBuilder;
import cryptix.message.LiteralMessage;
import cryptix.message.LiteralMessageBuilder;
import cryptix.message.Message;
import cryptix.message.MessageFactory;
import cryptix.openpgp.PGPArmouredMessage;
import cryptix.pki.KeyBundle;
import cryptix.message.EncryptedMessage;

public class PublicKeyEnc {

  /**
```

```
    *
    * @param p_msg
    * @param p_publicKey
    * @return
    * @throws Exception
    */
   public static String encrypt(String p_msg, KeyBundle
p_publicKey) throws Exception {
     LiteralMessage litmsg = null;
     LiteralMessageBuilder lmb =
LiteralMessageBuilder.getInstance("OpenPGP");
     lmb.init(p_msg);
     litmsg = (LiteralMessage) lmb.build();

     Message msg = null;
     EncryptedMessageBuilder emb =
EncryptedMessageBuilder.getInstance("OpenPGP");
     emb.init(litmsg);
     emb.addRecipient(p_publicKey);
     msg = emb.build();

     PGPArmouredMessage armoured;
     armoured = new PGPArmouredMessage(msg);
     ByteArrayOutputStream out = new ByteArrayOutputStream();
     out.write(armoured.getEncoded());
     out.close();
     return new String(out.toByteArray());
   }


   /**
    *
    * @param p_msg
    * @param p_secretKey
    * @param p_passwd
    * @return
    * @throws Exception
    */
   public static String decrypt(String p_msg, KeyBundle
p_secretKey, String p_passwd) throws Exception {

     MessageFactory mf = MessageFactory.getInstance("OpenPGP");
     ByteArrayInputStream in = new
ByteArrayInputStream(p_msg.getBytes());
     Collection msgs = mf.generateMessages(in);
     EncryptedMessage em = (EncryptedMessage)
msgs.iterator().next();
     in.close();
    Message msg = null;
    try {
      msg = em.decrypt(p_secretKey, p_passwd.toCharArray());
    } catch (Exception p_e) {
      p_e.printStackTrace();
      throw p_e;
    }
    return ((LiteralMessage) msg).getTextData();
  }

}
package scapi.kp.rsa;

import cryptix.openpgp.PGPKeyBundle;
import cryptix.pki.CertificateBuilder;
import cryptix.pki.PrincipalBuilder;
import cryptix.pki.KeyBundleFactory;
import java.security.Security;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Principal;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.cert.Certificate;
import cryptix.openpgp.PGPArmouredMessage;
import java.io.ByteArrayOutputStream;
import scapi.kp.IKeyPair;

public class RSAKeyPair implements IKeyPair {

  static final String rsa = "OpenPGP/Encryption/RSA";
  static final String elGamal = "OpenPGP/Signing/ElGamal";
  static final String dsa = "OpenPGP/Signing/DSA";

  protected String m_random = null;
  protected String m_passphrase = null;
  protected String m_username = null;
  protected PGPKeyBundle m_pgpPublicKey, m_pgpSecretKey;
  protected int m_signingAlgSize = 1024;
  protected int m_encryptAlgSize = 1024;

  static {
    Security.addProvider(new
cryptix.jce.provider.CryptixCrypto());
    Security.addProvider(new
cryptix.openpgp.provider.CryptixOpenPGP());
  }
  /**
   * Default consrtructor
   */
  public RSAKeyPair() {
  }

  /**
   * Constructor
   *
   * @param p_seed seed forrandom number generation
   * @param p_name user name
   * @param p_passphrase user password
   */
  public void genKeys(String p_seed, String p_name, String
p_passphrase) {
    m_random = p_seed;
    m_username = p_name;
    m_passphrase = p_passphrase;
    genKeys();
  }


  /**
   * Generate a key pair
   */
  private void genKeys(String signingAlg) {
    try {
      KeyBundleFactory kbf =
KeyBundleFactory.getInstance("OpenPGP");
      m_pgpPublicKey = (PGPKeyBundle)
kbf.generateEmptyKeyBundle();
      m_pgpSecretKey = (PGPKeyBundle)
kbf.generateEmptyKeyBundle();

      //generate the signature key
      KeyPairGenerator kpg =
KeyPairGenerator.getInstance(signingAlg);
      kpg.initialize(m_signingAlgSize, new
SecureRandom(m_random.getBytes()));
      KeyPair kp = kpg.generateKeyPair();
      PublicKey pubkey = kp.getPublic();
      PrivateKey privkey = kp.getPrivate();

      //add the certificate
      PrincipalBuilder princbuilder =
PrincipalBuilder.getInstance("OpenPGP/UserID");
      Principal userid = princbuilder.build(m_username);
      CertificateBuilder certbuilder =
CertificateBuilder.getInstance("OpenPGP/Self");
```

```
      Certificate cert = certbuilder.build(pubkey, userid,
privkey, new SecureRandom(m_random.getBytes()));
      m_pgpPublicKey.addCertificate(cert);
      m_pgpSecretKey.addCertificate(cert);

      m_pgpSecretKey.addPrivateKey(pubkey, privkey,
m_passphrase.toCharArray(), new
SecureRandom(m_random.getBytes()));

      //now generate the RSA key pair
      kpg = null;
      kpg = KeyPairGenerator.getInstance(rsaAlg);
      kpg.initialize(m_encryptAlgSize, new
SecureRandom(m_random.getBytes()));
      kp = kpg.generateKeyPair();
      PublicKey pubsubkey = kp.getPublic();
      PrivateKey privsubkey = kp.getPrivate();
      PublicKey pubmainkey = (PublicKey)
m_pgpSecretKey.getPublicKeys().next();
      PrivateKey privmainkey =
m_pgpSecretKey.getPrivateKey(pubmainkey,m_passphrase.toCharArra
y());
      m_pgpPublicKey.addPublicSubKey(pubsubkey, privmainkey);
      m_pgpSecretKey.addPublicSubKey(pubsubkey,
m_pgpPublicKey);
      m_pgpSecretKey.addPrivateSubKey(privsubkey, pubsubkey,
m_passphrase.toCharArray(), new
SecureRandom(m_random.getBytes()));
    } catch (Exception p_e) {
      p_e.printStackTrace();
    }
  }

  /**
   * to access the private key
   *
   * @return the armoured private key
   * @throws Exception
   */
  public String getArmouredPrivate() throws Exception {
    PGPArmouredMessage armoured = new
PGPArmouredMessage(m_pgpSecretKey);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    out.write(armoured.getEncoded());
    String res = new String(out.toByteArray());
    out.close();
    return res;
  }
  /**
   * to access the public key
   *
   * @return the public key
   * @throws Exception
   */
  public PGPKeyBundle getPublicKey() throws Exception {
    return m_pgpPublicKey;
  }

  /**
   * to access the public key
   *
   * @return the armoured public key
   * @throws Exception
   */
  public String getArmouredPublicKey() throws Exception {
    PGPArmouredMessage armoured = new
PGPArmouredMessage(m_pgpPublicKey);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    out.write(armoured.getEncoded());
    String res = new String (out.toByteArray());
    out.close();
    return res;
  }

  /**
   * to access the private key
   *
   * @return the private key
   * @throws Exception
   */
  public PGPKeyBundle getPrivateKey() throws Exception {
    return new
RSAPrivateKey(getArmouredPrivate()).getPrivateKey();
  }

}
```

APPENDIX C

SAMPLE SERVER CODE

```
package ssapi.pgp;

import java.util.Vector;

public interface ISecurityManager {

        public String savePublicKey(String p_user, String
p_armouredPublic) throws Exception;

        public String getPublicKey(String p_user) throws
Exception;

        public void deletePublicKey(String p_user) throws
Exception;

        public void updatePublicKey(String p_armouredPublic)
throws Exception;

        public void saveKeyLegitimacy(String p_user, String
p_trustee, int p_value) throws Exception;

        public KeyLegitimacy getKeyLegitimacy(String p_user,
String p_corr) throws Exception;

        public void deleteKeyLegitimacy(String p_user, String
p_trustee) throws Exception;

        public void updateKeyLegitimacy(String p_user, String
p_trustee, int p_value) throws Exception;

        public String saveKeys(String p_armouredPublic, String
p_armouredSecret) throws Exception;

        public void deleteKeyPair(String p_user) throws
Exception;

        public void updateKeyPair(String p_armouredPublic,
String p_armouredSecret) throws Exception;

        public String saveUserTrusts(Permission p_perm, Trust[]
p_trusts) throws Exception;

        public String getTrust(String p_user, String p_trustee)
throws Exception;

        public void deleteTrust(String p_user, String
p_trustee) throws Exception;
```

```
        public void updateTrust(String p_user, String
p_trustee) throws Exception;

        public String getSecretKey(String p_user) throws
Exception;

        public String getEmailAddressByPublicKeyId(String
p_KeyId) throws Exception;

        public String[] getPublicKeyOwners(String[] p_ids)
throws Exception;

        public void saveNewPublicKeySignatures(String p_keyId,
String[] p_signers) throws Exception;

        public String[] getPublicKeySignatures(String p_keyId)
throws Exception;

  public void saveNewPublicKeySignatures(String p_keyId, Vector
p_signatures) throws Exception;

}

package ssapi.pgp;

public class KeyLegitimacy {

  public int m_overAllLegitimacy;
  public int m_userLegitimacy;

}

package ssapi.pgp;

public class Trust {

  public String m_user;
  public String m_trusted;
  public int m_value;

  public String getM_user() {
    return m_user;
  }

  public String getM_trusted() {
    return m_trusted;
  }

  public int getM_value() {
    return m_value;
  }

  public void setM_user(String p_user) {
    m_user = p_user;
  }

  public void setM_trusted(String p_trusted) {
    m_trusted = p_trusted;
  }

  public void setM_value(int p_value) {
    m_value = p_value;
  }

}

package ssapi.pgp;

import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.util.Vector;
import java.util.LinkedList;
import java.util.Collection;
import java.security.PublicKey;
import java.security.cert.Certificate;
import cryptix.message.KeyBundleMessage;
import cryptix.message.MessageFactory;
import cryptix.pki.KeyBundle;

public class DefaultSecurityManagerImpl extends
SecurityManagerUtil implements ISecurityManager {

    static {
          try {
            java.security.Security.addProvider(new
cryptix.jce.provider.CryptixCrypto());
            java.security.Security.addProvider(new
cryptix.openpgp.provider.CryptixOpenPGP());
          } catch (Exception p_e) {
            p_e.printStackTrace();
            new SecurityManagerUtil().log(p_e);
          }
        }

        public void updatePublicKey(String p_armouredPublic)
throws Exception {
```

```
          Connection conn = null;
          Statement st = null;
          try {
            String firstName = null, lastName = null,
middleName = null, emailAddress = null;

            String keyOwner =
getPublicKeyOwner(p_armouredPublic);
            if (keyOwner == null)
              throw new Exception("Unable to get public key
owner.");
            StringTokenizer stt = new StringTokenizer(keyOwner,
" ");
            if (stt.countTokens() < 2)
              throw new Exception("Invalid public key owner.");
            firstName = stt.nextToken();
            middleName = stt.nextToken();
            if (!stt.hasMoreTokens()) { //2 tokens
              emailAddress = middleName;
              lastName = firstName;
              firstName = "";
            } else {
              lastName = stt.nextToken();
              if (!stt.hasMoreTokens()) { //3 tokens
                emailAddress = lastName;
                lastName = middleName;
                middleName = "";
              } else { //4 tokens
                emailAddress = stt.nextToken();
              }
            }

            if (emailAddress != null && emailAddress.length() >
2)
              emailAddress = emailAddress.substring(1,
emailAddress.length() - 1);

            String userid = getUserIdByEmail(emailAddress);

            String keyId =
getPublicKeyIdString(p_armouredPublic);
            KeyBundle bundle =
stringToBundle(p_armouredPublic);
            Vector signatures = new Vector();

            st = (conn = getDbConnection()).createStatement();
            Iterator itr = bundle.getCertificates();
            while (itr.hasNext()) {
              cryptix.openpgp.provider.PGPCertificateImpl cert
= (cryptix.openpgp.
                  provider.PGPCertificateImpl) itr.next();
              if (cert != null) {
                String issuerKeyId = toString( (
(cryptix.openpgp.PGPCertificate)
cert).getIssuerKeyID().getBytes());
                if (keyId.indexOf(issuerKeyId) < 0) {
                  signatures.add(cert);
                  String qry = "UPDATE INTO keysignatures (id,
user_id, key_id) VALUES (" +
                      System.currentTimeMillis() + ", '" +
getUserIdByKeyId(issuerKeyId) + "', '" + keyId + "')";
                  int count = st.executeUpdate(qry);
                  if (count <= 0) {
                    qry = "INSERT INTO keysignatures (id,
user_id, key_id) VALUES (" +
                      System.currentTimeMillis() + ", '" +
getUserIdByKeyId(issuerKeyId) + "', '" + keyId + "')";
                    st.executeUpdate(qry);
                  }
                }
              }
            }

            ResultSet rs = st.executeQuery("select count(*)
from publicKeys where key_id = '" + keyId + "'");
            if (rs.next()) {
              if (rs.getInt(1) <= 0) {
                String query = "UPDATE into publicKeys (id,
key_id, user_id, armoured_public) values (";
                query += getNextIndexVal("publickeys", "id") +
",'";
                query += keyId + "','";
                query += userid + "','";
                query += p_armouredPublic + "')";
                st.executeUpdate(query);
              }
            }
            //updatePublicKeySignatures(keyId, signatures);

          } catch (Exception p_e) {
            log(p_e);
            throw p_e;
          } finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
```

```
              p_e.printStackTrace();
            }
          if (conn != null)
            try {
              conn.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
        }
      }

      public void deleteKeyLegitimacy(String p_user, String
p_trustee) throws Exception {
        Connection conn = null;
        Statement st = null;
        try {
          st = (conn = getDbConnection()).createStatement();
          conn.setAutoCommit(false);

          String userid = getUserIdByEmail(p_user);
          String trusteeKeyId = getKeyIdByEmail(p_trustee);

          String sql = "DELETE FROM keylegitimacy WHERE
user_id = " + userid + " AND key_id = " + trusteeKeyId;
          System.out.println(sql);
          int count = st.executeUpdate(sql);

          conn.commit();
        } catch (Exception p_e) {
          conn.rollback();
          log(p_e);
          throw p_e;
        } finally {
          if (st != null)
            try {
              st.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
          if (conn != null)
            try {
              conn.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
        }
      }
```

```
        public void updateKeyLegitimacy(String p_user, String
p_trustee, int p_value) throws Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();
            conn.setAutoCommit(false);

            String userid = getUserIdByEmail(p_user);
            String trusteeKeyId = getKeyIdByEmail(p_trustee);

            String sql = "UPDATE keylegitimacy SET legitimacy =
" + p_value + " WHERE user_id = " + userid + " AND key_id = " +
trusteeKeyId;
            System.out.println(sql);
            int count = st.executeUpdate(sql);

            conn.commit();
          } catch (Exception p_e) {
            conn.rollback();
            log(p_e);
            throw p_e;
          } finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public String getTrust(String p_user, String p_trustee)
throws Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();

            String userid = getUserIdByEmail(p_user);
            String trusteeKeyId = getKeyIdByEmail(p_trustee);
            String sql = "SELECT legitimacy FROM keylegitimacy
WHERE userid = " + userid +  " AND key_id = " + trusteeKeyId;
            System.out.println(sql);
            ResultSet rs = st.executeQuery(sql);
            if (!rs.next())
              throw new Exception("Trust not found:" + sql);

            return rs.getString(1);

          } catch (Exception p_e) {
            log(p_e);
            throw p_e;
          } finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public void saveKeyLegitimacy(String
p_userEmailAddress, String p_correspondent, int p_value) throws
Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();

            String sql = "SELECT id FROM users WHERE
email_address='" +
                p_userEmailAddress + "'";
            System.out.println(sql);
            ResultSet rs = st.executeQuery(sql);
            if (!rs.next())
              throw new Exception("User Email address not
found.");
            String userid = rs.getString(1);

            sql = "SELECT id FROM users WHERE email_address='"
+ p_correspondent +
                "'";
            System.out.println(sql);
            rs = st.executeQuery(sql);
            if (!rs.next())
              throw new Exception("Trustee Email address not
found.");
            String trusteeId = rs.getString(1);

            sql = "SELECT id FROM publicKeys WHERE user_id='" +
trusteeId + "'";
            System.out.println(sql);
            rs = st.executeQuery(sql);
            if (!rs.next())
              throw new Exception("Email address not found.");
            String publicKeyId = rs.getString(1);

            sql =
                "INSERT INTO keyLegitimacy (id, user_id,
key_id, legitimacy) VALUES (" +
                System.currentTimeMillis() + "," + userid + ","
+ publicKeyId + "," +
                p_value + ")";
            System.out.println(sql);
            st.executeUpdate(sql);
          }
          catch (Exception p_e) {
            log(p_e);
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }

        }

        public String getKeyIdByEmail(String p_user) throws
Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();

            String query = "SELECT id FROM users WHERE
email_address = '" + p_user +
                "'";
            System.out.println(query);
            ResultSet rs = st.executeQuery(query);
            String userid = null;
            if (!rs.next())
              throw new Exception("Email address not found:" +
query);
            userid = rs.getString(1);

            query = "SELECT key_id FROM publickeys WHERE
user_id='" + userid + "'";
            System.out.println(query);
            rs = st.executeQuery(query);
            if (!rs.next())
              throw new Exception("User address not found:" +
query);
            return rs.getString(1);

          }
          catch (Exception p_e) {
            log(p_e);
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public String getEmailAddressByPublicKeyId(String
p_KeyId) throws Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();

            String query = "SELECT user_id FROM publicKeys
WHERE key_id like '%" +
                p_KeyId + "%'";
            System.out.println(query);
            ResultSet rs = st.executeQuery(query);
            String userid = null;
            if (!rs.next())
              throw new Exception("Email address not found:" +
query);
            userid = rs.getString(1);

            query = "SELECT email_address FROM users WHERE
id='" + userid + "'";
            System.out.println(query);
            rs = st.executeQuery(query);
            if (!rs.next())
              throw new Exception("User address not found:" +
query);
            return rs.getString(1);

          }
          catch (Exception p_e) {
            log(p_e);
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public String saveKeys(String p_armouredPublic, String
p_armouredSecret) throws
            Exception {
          Connection conn = null;
          Statement st = null;
```

```
         try {
         st = (conn = getDbConnection()).createStatement();

         String newUserId = "" + System.currentTimeMillis();
         String firstName = null, lastName = null, middleName
= null, emailAddress = null;

         System.out.println("Key owner: " +
getPublicKeyOwner(p_armouredPublic));
         String keyOwner =
getPublicKeyOwner(p_armouredPublic);
         if (keyOwner == null)
           throw new Exception("Unable to get public key
ownoer.");
         java.util.StringTokenizer stt = new
java.util.StringTokenizer(keyOwner, " ");
         if (stt.countTokens() < 2)
           throw new Exception("Invalid public key owner.");
         firstName = stt.nextToken();
         middleName = stt.nextToken();
         if (!stt.hasMoreTokens()) { //2 tokens
           emailAddress = middleName;
           lastName = firstName;
           firstName = "";
         }
         else {
           lastName = stt.nextToken();
           if (!stt.hasMoreTokens()) { //3 tokens
             emailAddress = lastName;
             lastName = middleName;
             middleName = "";
           }
           else { //4 tokens
             emailAddress = stt.nextToken();
           }
         }

         if (emailAddress != null && emailAddress.length() >
2)
           emailAddress = emailAddress.substring(1,
emailAddress.length() - 1);

         String query = "insert into users (id, first_name,
middle_name, last_name, email_address) values ('";
         query += newUserId + "','";
         query += firstName + "','";
         query += middleName + "','";
         query += lastName + "','";
         query += emailAddress + "')";
```

```
          log("query:" + query);
          st.executeUpdate(query);

          query = "insert into privateKeys (user_id, key_id,
armoured_private, armoured_public) values ('";
          query += newUserId + "','";
          query += getPublicKeyIdString(p_armouredPublic) +
"','";
          query += p_armouredSecret + "','";
          query += p_armouredPublic + "')";
          st.executeUpdate(query);

          st.close();

          savePublicKey(emailAddress, p_armouredPublic);
        }
        catch (Exception p_e) {
          log(p_e);
          throw p_e;
        }
        finally {
          if (st != null)
            try {
              st.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
          if (conn != null)
            try {
              conn.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
        }

          return "done.";
        }

        public void deleteKeyPair(String p_emailAddress) throws
Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();

            System.out.println("SELECT id FROM users WHERE
email_address='" +
                                   p_emailAddress + "'");
            ResultSet rs = st.executeQuery(
                "SELECT id FROM users WHERE email_address='" +
p_emailAddress + "'");
            String userid = null;
            if (!rs.next())
              throw new Exception("Email address not found.");
            userid = rs.getString(1);

            System.out.println("DELETE FROM privateKeys WHERE
user_id = " + userid);
            st.executeUpdate("DELETE FROM privateKeys WHERE
user_id = " + userid);

            System.out.println("DELETE FROM users WHERE id = "
+ userid);
            st.executeUpdate("DELETE FROM users WHERE id = " +
userid);
          }
          catch (Exception p_e) {
            log(p_e);
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public String savePublicKey(String p_user, String
p_armouredPublic) throws Exception {
          Connection conn = null;
          Statement st = null;
          try {
            String firstName = null, lastName = null,
middleName = null, emailAddress = null;

            System.out.println("Key owner:" +
getPublicKeyOwner(p_armouredPublic));
            String keyOwner =
getPublicKeyOwner(p_armouredPublic);
```

```
           if (keyOwner == null)
             throw new Exception("Unable to get public key
ownoer.");
           StringTokenizer stt = new StringTokenizer(keyOwner,
" ");
           if (stt.countTokens() < 2)
             throw new Exception("Invalid public key owner.");
           firstName = stt.nextToken();
           middleName = stt.nextToken();
           if (!stt.hasMoreTokens()) { //2 tokens
             emailAddress = middleName;
             lastName = firstName;
             firstName = "";
           } else {
             lastName = stt.nextToken();
             if (!stt.hasMoreTokens()) { //3 tokens
               emailAddress = lastName;
               lastName = middleName;
               middleName = "";
             } else { //4 tokens
               emailAddress = stt.nextToken();
             }
           }

           if (emailAddress != null && emailAddress.length() >
2)
             emailAddress = emailAddress.substring(1,
emailAddress.length() - 1);

           st = (conn = getDbConnection()).createStatement();
           ResultSet rs = st.executeQuery(
               "SELECT id FROM users WHERE email_address = '"
+ emailAddress + "'");
           String userid = null;
           if (rs.next())
             userid = rs.getString(1);
           else {
             String newUserId = userid = "" +
System.currentTimeMillis();
             String query = "insert into users (id,
first_name, middle_name, last_name, email_address) values ('";
             query += newUserId + "','";
             query += firstName + "','";
             query += middleName + "','";
             query += lastName + "','";
             query += emailAddress + "')";
             log("query:" + query);
             st.executeUpdate(query);
           }
```

```
            String keyId =
getPublicKeyIdString(p_armouredPublic);
            KeyBundle bundle =
stringToBundle(p_armouredPublic);
            Vector signatures = new Vector();

            Iterator itr = bundle.getCertificates();
            while (itr.hasNext()) {
              cryptix.openpgp.provider.PGPCertificateImpl cert
= (cryptix.openpgp.
                    provider.PGPCertificateImpl) itr.next();
              if (cert != null) {
                String issuerKeyId = toString( (
(cryptix.openpgp.PGPCertificate)

cert).getIssuerKeyID().getBytes());
                if (keyId.indexOf(issuerKeyId) < 0) {
                  signatures.add(cert);
                  String qry =
                      "INSERT INTO keysignatures (id, user_id,
key_id) VALUES (" +
                      System.currentTimeMillis() + ", '" +
                      getUserIdByKeyId(issuerKeyId) + "', '" +
keyId + "')";
                  System.out.println(qry);
                  st.executeUpdate(qry);
                }
              }
            }

            rs = st.executeQuery("select count(*) from
publicKeys where key_id = '" + keyId + "'");
            if (rs.next()) {
              if (rs.getInt(1) <= 0) {
                String query = "insert into publicKeys (id,
key_id, user_id, armoured_public) values (";
                query += getNextIndexVal("publickeys", "id") +
",'";
                query += keyId + "','";
                query += userid + "','";
                query += p_armouredPublic + "')";
                log(query);
                st.executeUpdate(query);
              }
            }
            saveNewPublicKeySignatures(keyId, signatures);

            return "Key has been saved for " + p_user;
```

```
        } catch (Exception p_e) {
          log(p_e);
          throw p_e;
        } finally {
          if (st != null)
            try {
              st.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
          if (conn != null)
            try {
              conn.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
        }
      }

      public String getPublicKey(String p_emailAddress)
throws Exception {
        if (p_emailAddress == null ||
p_emailAddress.equals(""))
          throw new Exception("Invalid email address.");

        Connection conn = null;
        Statement st = null;
        try {
          st = (conn = getDbConnection()).createStatement();

          String userid = getUserIdByEmail(p_emailAddress);
          ResultSet rs = st.executeQuery(
              "select armoured_public from privateKeys where
user_id = '" + userid +
              "'");
          if (rs.next())
            return rs.getString(1);

          rs = st.executeQuery(
              "select armoured_public from publicKeys where
user_id = '" + userid +
              "'");
          if (rs.next())
            return rs.getString(1);
          else
            throw new Exception("Public key not found for " +
p_emailAddress);

        }
```

```
          catch (Exception p_e) {
            p_e.printStackTrace();
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public void deletePublicKey(String p_emailAddress)
throws Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();
            conn.setAutoCommit(false);

            String userid = getUserIdByEmail(p_emailAddress);

            String sql = "DELETE FROM publicKeys WHERE user_id
= " + userid;
            System.out.println(sql);
            int count = st.executeUpdate(sql);
            if (count < 1)
              throw new Exception("Public key not found.");

            sql = "DELETE FROM trust WHERE user_id = " +
userid;
            System.out.println(sql);
            count = st.executeUpdate(sql);

            sql = "DELETE FROM keylegitimacy WHERE user_id = "
+ userid;
            System.out.println(sql);
            count = st.executeUpdate(sql);

            conn.commit();
          } catch (Exception p_e) {
            //conn.rollback();
            log(p_e);
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public String getSecretKey(String p_emailAddress)
throws Exception {
          if (p_emailAddress == null ||
p_emailAddress.equals(""))
            throw new Exception("Invalid email address.");

          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();

            ResultSet rs = st.executeQuery(
                "SELECT id FROM users WHERE email_address = '"
+ p_emailAddress + "'");
            String userid = null;
            if (rs.next())
              userid = rs.getString(1);
            else
              throw new Exception("E-mail address not found.");
            rs = st.executeQuery(
                "SELECT armoured_private FROM privateKeys WHERE
user_id = '" + userid +
                "'");
            if (rs.next())
              return rs.getString(1);
            return null;
          }
          catch (Exception p_e) {
            log(p_e);
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public String[] getPublicKeyOwners(String[] p_ids)
throws Exception {
          Connection conn = null;
          Statement st = null;
          try {
            if (p_ids == null)
              return null;
            if (p_ids.length <= 0)
              return new String[] {};
            String list = "";
            for (int i = 0; i < p_ids.length; i++) {
              list += "key_id like '%" + p_ids[i] + "%'";
              if (i < p_ids.length - 1)
                list += " OR ";
            }

            st = (conn = getDbConnection()).createStatement();

            ResultSet rs = st.executeQuery("select user_id from
publicKeys where " +
                                              list);
            LinkedList resList = new LinkedList();
            while (rs.next())
              resList.add(rs.getString(1));

            rs = st.executeQuery("select user_id from
privateKeys where " + list);
            while (rs.next())
              resList.add(rs.getString(1));

            String[] res = new String[resList.size()];
          for (int i = 0; i < resList.size(); i++)
            res[i] = (String) resList.get(i);
          return res;
        }
        catch (Exception p_e) {
          log(p_e);
          throw p_e;
        }
        finally {
          if (st != null)
            try {
              st.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
          if (conn != null)
            try {
              conn.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
        }
      }

      public void deleteTrust(String p_userEmailAddress,
                              String p_trusteeEmailAddress)
throws Exception {
        Connection conn = null;
        Statement st = null;
        try {
          st = (conn = getDbConnection()).createStatement();

          ResultSet rs = st.executeQuery(
              "SELECT id FROM users WHERE email_address='" +
p_userEmailAddress +
              "'");
          String userid = null;
          if (!rs.next())
            throw new Exception("Email address not found.");
          userid = rs.getString(1);

          rs = st.executeQuery("SELECT id FROM users WHERE
email_address='" +
                               p_trusteeEmailAddress + "'");
          String trustee = null;
          if (!rs.next())
            throw new Exception("Email address not found.");
          trustee = rs.getString(1);
```

```
            rs = st.executeQuery("SELECT key_id FROM publicKeys
WHERE user_id='" +
                                    trustee + "'");
            String trusteeKeyId = null;
            if (!rs.next())
              throw new Exception("Email address not found.");
            trusteeKeyId = rs.getString(1);

            st.executeUpdate("DELETE FROM trust WHERE user_id =
" + userid +
                                " AND trustee_public_key_id = '" +
trusteeKeyId + "'");

          }
          catch (Exception p_e) {
            log(p_e);
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
        }

        public void deleteUser(String p_emailAddress) throws
Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();

            st.executeUpdate("DELETE FROM users WHERE
email_address = '" + p_emailAddress + "'");

          } catch (Exception p_e) {
            log(p_e);
            throw p_e;
          } finally {
            if (st != null)
```

```
                try {
                  st.close();
                } catch (Exception p_e) {
                  p_e.printStackTrace();
                }
              if (conn != null)
                try {
                  conn.close();
                } catch (Exception p_e) {
                  p_e.printStackTrace();
                }
            }
          }

          public String saveUserTrusts(Permission p_perm, Trust[]
p_trusts) throws
              Exception {
            Connection conn = null;
            Statement st = null;
            try {
              if (p_trusts == null)
                return "Null Array.";
              if (p_trusts.length <= 0)
                return "Array empty.";
              for (int i = 0; i < p_trusts.length; i++) {
                String trusterId =
getUserIdByEmail(p_trusts[i].m_user);
                String trusteeKeyId =
getUserIdByEmail(p_trusts[i].m_trusted);
                st = (conn =
getDbConnection()).createStatement();
                st.executeUpdate("DELETE FROM trust WHERE
user_id= '" +
                                 p_trusts[i].m_user + "' AND
trustee_public_key_id='" +
                                 p_trusts[i].m_trusted + "'");
                String query =
                    "INSERT INTO trust (id, user_id,
trustee_public_key_id, level) VALUES ";
                query += "(" + getNextIndexVal("trust", "id") +
",'" +
                    trusterId + "','" + trusteeKeyId + "'," +
                    p_trusts[i].m_value + ")";
                st = conn.createStatement();
                st.executeUpdate(query);
                st.close();
              }
            }
            catch (Exception ex) {
```

```
            log(ex);
            throw ex;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
               p_e.printStackTrace();
              }
            if (conn != null)
              try {
                conn.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          }
          return "";
        }

        public String[] getPublicKeySignatures(String p_keyId)
throws Exception {
          Connection conn = null;
          Statement st = null;
          try {
            st = (conn = getDbConnection()).createStatement();
            String query = "SELECT * FROM trust WHERE
trustee_public_key_id LIKE '%";
            query += p_keyId + "%'";
            ResultSet rs = st.executeQuery(query);
            Vector signatures = new Vector();
            while (rs.next())
              signatures.add(rs.getString("user"));
            String[] res = new String[signatures.size()];
            for (int i = 0; i < signatures.size(); i++)
              res[i] = (String) signatures.get(i);
            return res;
          }
          catch (Exception p_e) {
            log(p_e);
            throw p_e;
          }
          finally {
            if (st != null)
              try {
                st.close();
              } catch (Exception p_e) {
                p_e.printStackTrace();
              }
          if (conn != null)
            try {
              conn.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
        }
      }

      public synchronized void
saveNewPublicKeySignatures(String p_keyId,
          String[] p_signers) throws Exception {
        Connection conn = null;
        Statement st = null;
        try {
          st = (conn = getDbConnection()).createStatement();
          String query = "";
          if (p_signers.length > 0) {
            for (int i = 0; i < p_signers.length; i++) {
              query = "insert into trust (id, user, trusts)
values ";
              long nextVal = getNextIndexVal("trust", "id");
              query += " (" + nextVal++ +",'";
              query += p_signers[i] + "','"; //put the ketid
istead of the email address
              query += p_keyId + "')";
              st = conn.createStatement();
              st.executeUpdate(query);
            }
          }
        }
        catch (Exception p_e) {
          log(p_e);
          throw p_e;
        }
        finally {
          if (st != null)
            try {
              st.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
          if (conn != null)
            try {
              conn.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
        }
        }

        public synchronized void
saveNewPublicKeySignatures(String p_keyId,
            Vector p_signatures) throws Exception {
          try {
            String[] signatures =
getPublicKeySignatures(p_keyId);
            String signaturesList = "";
            for (int i = 0; i < signatures.length; i++)
              signaturesList += signatures[i] + ",";
              //Remove already exiting or self signtures
            for (int i = 0; i < p_signatures.size(); i++) {
              Certificate cert = (Certificate)
p_signatures.get(i);
              String strKeyId = getPublicKeyIdString( (
(cryptix.openpgp.
                  PGPCertificate) cert).getPublicKey());
              if (signaturesList.indexOf(strKeyId) >= 0 ||
                  strKeyId.indexOf(p_keyId) >= 0)
                p_signatures.remove(i--);
            }

            String[] signers = new String[p_signatures.size()];
            if (p_signatures.size() > 0) {
              for (int i = 0; i < p_signatures.size(); i++) {
                Certificate cert = (Certificate)
p_signatures.get(i);
                String strKeyId = toString( (
(cryptix.openpgp.PGPCertificate) cert).

getIssuerKeyID().getBytes());
                signers[i] = strKeyId;
              }
              saveNewPublicKeySignatures(p_keyId, signers);
            }
          }
          catch (Exception p_e) {
            log(p_e);
            throw p_e;
          }
        }

        public KeyLegitimacy getKeyLegitimacy(String p_user,
String p_corr) throws
            Exception {
          KeyLegitimacy keyLegitimacy = new KeyLegitimacy();
          keyLegitimacy.m_userLegitimacy =
keyLegitimacy.m_overAllLegitimacy = 0;
```

```
/*
 G - arbitrary connected graph
 v0 - is the initial beginning node
 V - is the set of all vertices in the graph G
 S - set of all vertices with permanent labels
 n - number of vertices in G
 D - set of distances to v0
 C - set of edges in G
 Dijkstra (graph G, node v0) {
 S={v0}
 For i = 1 to n
   D[i] = C[v0,i]
 For i = 1 to n-1 {
   Choose node w in V-S with min D[w]
   Add w to S
   For each node v in V-S
     D[v] = min(D[v], D[w] + C[w,v])
 }
 }
 */
try {
  //validate the user
  String userid = getUserIdByEmail(p_user);
  String corrid = getUserIdByEmail(p_corr);
  Connection conn = null;
  Statement st = null;
  try {
    st = (conn =
getDbConnection()).createStatement();
    ResultSet rs = st.executeQuery(
        "select id from publicKeys where user_id = '"
+ corrid + "'");
    if (!rs.next())
      throw new Exception("Public key not found.");

    String keyId = rs.getString(1);
    rs = st.executeQuery(
        "SELECT legitimacy FROM keyLegitimacy WHERE
user_id = '" + userid +
        "' AND key_id = '" + keyId + "'");
    if (rs.next())
      keyLegitimacy.m_userLegitimacy = rs.getInt(1);
    else
      keyLegitimacy.m_userLegitimacy = 0;

  } catch (Exception p_e) {
    log(p_e);
    throw (p_e);
```

```
            } finally {
              if (st != null)
                try {
                  st.close();
                } catch (Exception p_e) {
                  p_e.printStackTrace();
                }
              if (conn != null)
                try {
                  conn.close();
                } catch (Exception p_e) {
                  p_e.printStackTrace();
                }
            }

            MyDijkstra dijkstra = new MyDijkstra();
            dijkstra.runDijkstra(getGraph(p_user, p_corr),
p_user);

            keyLegitimacy.m_overAllLegitimacy = (int)
(Math.round(Double.parseDouble( (
                String) dijkstra.S.get(p_corr))));
            dijkstra.S = new Hashtable();
          }
          catch (Exception ex) {
            ex.printStackTrace();
          }

          return keyLegitimacy;
        }

        public Hashtable getGraph(String p_source, String
p_dest) throws Exception {
          Connection conn = null;
          Statement st = null;
          Hashtable predecessors = new Hashtable();
          Vector nodes = new Vector();
          nodes.add(p_dest);
          try {
            st = (conn = getDbConnection()).createStatement();

            ResultSet rs = st.executeQuery("SELECT * FROM
keysignatures, users WHERE keysignatures.user_id=users.id AND
key_id = '" +

getKeyIdByEmail(p_dest) + "'");
            Vector signers = new Vector();
            while (rs.next()) {
              signers.add(new Object[]
{rs.getString("email_address"), "100"});
              nodes.add(rs.getString("email_address"));
            }

            Object[] signersArray = new Object[signers.size()];
            for (int i = 0; i < signers.size(); i++)
              signersArray[i] = signers.get(i);

            predecessors.put(p_dest, signersArray);

            TreeSet processed = new TreeSet();
            processed.add(p_dest);

            int counter = 0;
            while (true) {
              if (counter++ >= nodes.size())
                break;

              String current = (String) nodes.get(counter - 1);
              if (!processed.contains(current)) {

                rs = st.executeQuery("SELECT * FROM trust,
users WHERE trust.user_id=users.id AND trustee_public_key_id =
'" +
                                              getUserIdByEmail(current)
+ "'");
                signers = new Vector();
                while (rs.next()) {
                  signers.add(new Object[]
{rs.getString("email_address"),
                                    rs.getString("level")});
                  nodes.add(rs.getString("email_address"));
                }

                signersArray = new Object[signers.size()];
                for (int i = 0; i < signers.size(); i++)
                  signersArray[i] = signers.get(i);
                predecessors.put(current, signersArray);

                processed.add(current);
              }

            }
          } catch (Exception p_e) {
            log(p_e);
            throw (p_e);
          } finally {
            if (st != null)
            try {
              st.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
          if (conn != null)
            try {
              conn.close();
            } catch (Exception p_e) {
              p_e.printStackTrace();
            }
        }

        return predecessors;
      }

      protected Challenge getNonce() throws Exception {
        Challenge challenge = new Challenge();
        challenge.m_nonce = "" + Math.round(Math.random() *
10000000000L);
        return challenge;
  }


}
```

APPENDIX D

SAMPLE E-MAIL APPLICATION CODE

```
package emailclient;

import java.io.InputStream;
import java.util.Properties;
import java.awt.BorderLayout;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Dimension;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseMotionListener;
import scapi.pgp.PGPServerConnector;
import netscape.javascript.JSObject;
import javax.swing.*;

public class NewAccountApplet extends JApplet implements
ActionListener {

  protected JTextField m_login;
  protected JPasswordField m_pgpPassword;
  protected JPasswordField m_accountPassword;
  protected JTextField m_incomingPopServer;
  protected JTextField m_popUser;
  protected JTextField m_popPasswd;
  protected JTextField m_fullName;
  protected String m_random;
  protected String m_strPublicKey;
  protected JButton m_submitButton = null;

  protected static String m_serverUrl;

  static {
    try {
      InputStream is =
Class.forName("scapi.util.ServerConnector").getResourceAsStream
("SecurityManager.properties");
      Properties props = new Properties();
      props.load(is);
      m_serverUrl = (String) props.get("SecurityManager");
      is.close();
    } catch (Exception p_e) {
      p_e.printStackTrace();
    }
  }

  public void init() {
    JPanel contentPane = new JPanel();
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();
    contentPane.setLayout(gridbag);
    c.fill = GridBagConstraints.HORIZONTAL;

    JLabel amountLabel = new JLabel("User full Name:");
    c.gridx = 0;
    c.gridy = 0;
    gridbag.setConstraints(amountLabel, c);
    contentPane.add(amountLabel);

    m_fullName = new JTextField(25);
    m_fullName.setText("Tawfik Lachheb");
    c.gridx = 1;
    c.gridy = 0;
    gridbag.setConstraints(m_fullName, c);
    contentPane.add(m_fullName);

    amountLabel = new JLabel("User login:");
    c.gridx = 0;
    c.gridy = 1;
    gridbag.setConstraints(amountLabel, c);
    contentPane.add(amountLabel);

    m_login = new JTextField(25);
    m_login.setText("tawfik");
    c.gridx = 1;
    c.gridy = 1;
    gridbag.setConstraints(m_login, c);
    contentPane.add(m_login);

    JLabel accountPasswordLabel = new JLabel("Email acount
password: ");
    c.gridx = 0;
    c.gridy = 2;
    gridbag.setConstraints(accountPasswordLabel, c);
    contentPane.add(accountPasswordLabel);

    m_accountPassword = new JPasswordField(25);
    m_accountPassword.setText("lachheb");
    c.gridwidth = 2;
    c.gridx = 1;
    c.gridy = 2;
    gridbag.setConstraints(m_accountPassword, c);
    contentPane.add(m_accountPassword);

    JLabel pgpPasswordLabel = new JLabel("PGP password:");
    c.gridx = 0;
    c.gridy = 3;
```

```
gridbag.setConstraints(pgpPasswordLabel, c);
contentPane.add(pgpPasswordLabel);

m_pgpPassword = new JPasswordField(25);
m_pgpPassword.setText("lachheb");
c.gridwidth = 2;
c.gridx = 1;
c.gridy = 3;
gridbag.setConstraints(m_pgpPassword, c);
contentPane.add(m_pgpPassword);

JLabel popLabel = new JLabel("POP3 server:");
c.gridx = 0;
c.gridy = 4;
gridbag.setConstraints(popLabel, c);
contentPane.add(popLabel);

m_incomingPopServer = new JTextField("mail.cula.net");
c.gridx = 1;
c.gridy = 4;
gridbag.setConstraints(m_incomingPopServer, c);
contentPane.add(m_incomingPopServer);

JLabel popUserLabel = new JLabel("POP3 user:");
c.gridx = 0;
c.gridy = 5;
gridbag.setConstraints(popUserLabel, c);
contentPane.add(popUserLabel);

m_popUser = new JTextField("tawfik");
c.gridx = 1;
c.gridy = 5;
gridbag.setConstraints(m_popUser, c);
contentPane.add(m_popUser);

JLabel popPasswdLabel = new JLabel("POP3 password:");
c.gridx = 0;
c.gridy = 6;
gridbag.setConstraints(popPasswdLabel, c);
contentPane.add(popPasswdLabel);

m_popPasswd = new JTextField("lachheb");
c.gridx = 1;
c.gridy = 6;
gridbag.setConstraints(m_popPasswd, c);
contentPane.add(m_popPasswd);

m_submitButton = new JButton("Submit");
m_submitButton.addActionListener(this);
```

```
    c.weighty = 1.0;
    c.gridwidth = 1;
    c.gridx = 1;
    c.gridy = 7;
    gridbag.setConstraints(m_submitButton, c);
    contentPane.add(m_submitButton);

    contentPane.setBackground(new Color(255,255,255));
    setContentPane(contentPane);
  }

  public void actionPerformed(ActionEvent e) {
    KeyGenMouseApplet keyGenMouseApplet = new
KeyGenMouseApplet();
    keyGenMouseApplet.m_accountApplet = this;
    JLabel emptyLabel = new JLabel("  Move the mouse until the
window closes.");
    emptyLabel.setPreferredSize(new Dimension(300, 300));
    keyGenMouseApplet.getContentPane().add(emptyLabel,
BorderLayout.CENTER);
    keyGenMouseApplet.pack();
    keyGenMouseApplet.setVisible(true);
  }

  public void confirmAccount() {
    JPanel contentPane = new JPanel();
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();
    contentPane.setLayout(gridbag);
    c.fill = GridBagConstraints.HORIZONTAL;

    String msg = "Your account has been created, here is your
public key:";
    JLabel amountLabel = new JLabel(msg);
    c.gridx = 0;
    c.gridy = 0;
    gridbag.setConstraints(amountLabel, c);
    contentPane.add(amountLabel);

    JTextArea body = new JTextArea(10,12);
    c.gridx = 0;
    c.gridy = 1;
    gridbag.setConstraints(body, c);
    body.setText(m_strPublicKey);
    contentPane.add(body);
    javax.swing.JScrollPane sc = new
javax.swing.JScrollPane(contentPane);
    setContentPane(sc);
  }
  public void genKeys() {
    try {
      String name = m_login.getText();
      scapi.kp.IKeyPair keyPair =
scapi.kp.KeyPairFactory.getInstance("RSA");
      keyPair.genKeys(m_random, m_login.getText(),
m_pgpPassword.getText());
      m_strPublicKey = keyPair.getArmouredPublicKey();
      java.util.Hashtable params = new java.util.Hashtable();
      params.put("cmd", "createAccount");
      params.put("user", m_login.getText());
      java.security.MessageDigest md =
java.security.MessageDigest.getInstance("SHA1",
"CryptixCrypto");
      String hashedPasswd = new
String(md.digest(m_accountPassword.getText().getBytes()));
      params.put("password", new
String(toString(hashedPasswd.getBytes())));
      params.put("popserver", m_incomingPopServer.getText());
      params.put("popusr", m_popUser.getText());
      params.put("popfullname",
java.net.URLEncoder.encode(m_fullName.getText()));
      params.put("poppasswd", m_popPasswd.getText());
      new
scapi.util.ServerConnector("http://"+EmailAppletUtil.getMailSer
verUrl((String) ( (JSObject)
JSObject.getWindow(this).getMember("location")).getMember("host
"))).sendHTTPGetMessage("", params);
      PGPServerConnector.setBaseUrl("http://"+(String) (
(JSObject)
JSObject.getWindow(this).getMember("location")).getMember("host
"));
      System.out.println("Sending new keys.");
      PGPServerConnector.sendKeyPair(keyPair,
m_fullName.getText()+"
<"+m_popUser.getText()+"@"+m_incomingPopServer.getText()+">");
      m_submitButton.setVisible(false);
    } catch (Exception p_e) {
      p_e.printStackTrace();
    }
  }

  class KeyGenMouseApplet extends JFrame implements
MouseMotionListener, ActionListener {

    public NewAccountApplet m_accountApplet;
    private long randomNumber = 0;
    private int count = 0, prevX = 0, prevY = 0;
    private String m_randomString = "";

    public void init(Graphics g) {
      setBackground(Color.blue);
      g.setColor(Color.red);
      g.setColor(Color.white);
      g.drawString("  Move the mouse on this window until it
closes.", 10, 15);
    }

    public KeyGenMouseApplet() {
      addMouseMotionListener(this);
    }

    public void mouseMoved(java.awt.event.MouseEvent me) {
      int x = me.getX();
      int y = me.getY();
      if (Math.abs(x - prevX) > 10 && Math.abs(y - prevY) > 10)
{
        count++;
        if (count % 5 == 0) {
          long temp=System.currentTimeMillis()%1000000000;
          temp = temp * x * y;
          temp = temp % 111222333;
          randomNumber += temp;
          m_randomString += randomNumber;
          System.out.println(randomNumber);
        }
      }
      if ((count % 150 == 140) && (m_randomString.length() >
256)) {
        try {
          Class.forName("scapi.pgp.PGPServerConnector");//load
the provider
          java.security.MessageDigest md =
java.security.MessageDigest.
              getInstance("SHA1", "CryptixCrypto");
           m_randomString = new
String(md.digest(m_randomString.getBytes()));
        } catch (Exception p_e) {
          p_e.printStackTrace();
        }
        m_accountApplet.m_random = m_randomString;
        setVisible(false);
        m_accountApplet.genKeys();
        m_accountApplet.confirmAccount();
        dispose();
      }
      prevX = x;
      prevY = y;
    }

    public void mouseDragged(java.awt.event.MouseEvent me) { }
    public void actionPerformed(java.awt.event.ActionEvent ae)
{ }

  }

  static byte[] hexFromString(String hex)
  {
      int len = hex.length();
      byte[] buf = new byte[((len + 1) / 2)];

      int i = 0, j = 0;
      if ((len % 2) == 1)
          buf[j++] = (byte) fromDigit(hex.charAt(i++));

      while (i < len) {
          buf[j++] = (byte) ((fromDigit(hex.charAt(i++)) << 4)
| fromDigit(hex.charAt(i++)));
      }
      return buf;
  }


  /**
   * Returns the number from 0 to 15 corresponding to the hex
digit <i>ch</i>.
   */
  static int fromDigit(char ch)
  {
      if (ch >= '0' && ch <= '9')
          return ch - '0';
      if (ch >= 'A' && ch <= 'F')
          return ch - 'A' + 10;
      if (ch >= 'a' && ch <= 'f')
          return ch - 'a' + 10;

      throw new IllegalArgumentException("invalid hex digit '"
+ ch + "'");
  }


  /**
   * Compares two byte arrays for equality.
   *
   * @return true if the arrays have identical contents
   */
  static boolean areEqual (byte[] a, byte[] b)
  {
      int aLength = a.length;
      if (aLength != b.length)
          return false;
      for (int i = 0; i < aLength; i++)
          if (a[i] != b[i])
              return false;
      return true;
  }


  /**
   * Returns a string of hexadecimal digits from a byte array.
Each
   * byte is converted to 2 hex symbols.
   */
  static String toString( byte[] ba )
  {
      int length = ba.length;
      char[] buf = new char[length * 2];
      for (int i = 0, j = 0, k; i < length; )
      {
          k = ba[i++];
          buf[j++] = HEX_DIGITS[(k >>> 4) & 0x0F];
          buf[j++] = HEX_DIGITS[ k        & 0x0F];
      }
      return new String(buf);
  }

  private static final char[] HEX_DIGITS =
  {

'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'
  };

}

      package emailserver;

      public class ActionFactory {

        public static IAction getInstance(String p_cmd) {
          if (p_cmd.equals("login"))
            return new LoginAction();
          if (p_cmd.equals("checkMailbox"))
            return new GetEmailsAction();
          if (p_cmd.equals("createAccount"))
            return new CreateAccountAction();
          if (p_cmd.equalsIgnoreCase("getEmailBody"))
            return new GetEmailAction();
          if (p_cmd.equalsIgnoreCase("sendEmail"))
            return new SendEmailAction();

          return null;
        }

}

package emailserver;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.io.PrintWriter;
import java.net.URLDecoder;

public class CreateAccountAction extends AAction {

  public void process(HttpServletRequest p_req,
HttpServletResponse p_resp) throws Exception {
    PrintWriter pw = p_resp.getWriter();
    String query = "insert into emailusers (id, userlogin,
userpasswd, popaccount, popuser, poppasswd, username,
popserver) values ('";
    query += System.currentTimeMillis() + "','";
    query += getParamQueryString(p_req, "user") + "','";
    query += URLDecoder.decode(getParamQueryString(p_req,
"password")) + "','";
    query += getParamQueryString(p_req, "popusr") + "','";
    query += getParamQueryString(p_req, "popusr") + "','";
    query += getParamQueryString(p_req, "poppasswd") + "','";
    query += URLDecoder.decode(getParamQueryString(p_req,
"popfullname")) + "','";
    query += getParamQueryString(p_req, "popserver") + "')";

    System.out.println("query: "+query);
    Connection conn =
DriverManager.getConnection(AAction.m_jdbcUrl, "tawfik",
"lachheb");
    Statement st = conn.createStatement();
    st.executeUpdate(query);
    st.close();
    conn.close();

    pw.print("Your account was created.");
    pw.close();
  }

}
```

APPENDIX E
SERVICE DEPLOYMENT DESCRIPTOR

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java" >
      <service name="SecurityManagerService"
provider="java:RPC" xmlns:myNS="urn:SecurityManagerService" >
                 <parameter name="className"
value="ssapi.pgp.SecurityManagerService" />
                 <parameter name="allowedMethods" value="*" />
                 <beanMapping qname="myNS:Permission"
languageSpecificType="java:ssapi.pgp.Permission" />
                 <beanMapping qname="myNS:Trust"
languageSpecificType="java:ssapi.pgp.Trust" />
                 <beanMapping qname="myNS:KeyLegitimacy"
languageSpecificType="java:ssapi.pgp.KeyLegitimacy" />
                 <beanMapping qname="myNS:Challenge"
languageSpecificType="java:ssapi.pgp.Challenge" />
           </service>
</deployment>
```

REFERENCES

[1] Gong, L., Ellison, G., & Dageforde, A., Inside Java 2 Platform Security: Architecture, API Design, and Implementation. Addison-Wesley Pub Co, second edition, May 2003.

[2] Stallings, W., Cryptography and Network Security. Prentice Hall, third edition, August 2002.

[3] Oaks, S., Java Security. O'Reilly & Associates, Inc., second edition, May 2001.

[4] Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A., Handbook of Applied Cryptography. CRC Press, October 1996.

[5] Sun Microsystems, Java Cryptography Architecture API Specification & Reference, (http://java.sun.com/j2se/1.4/docs/guide/security/HowToImplAProvider.html). Last updated in February 2002.

[6] Sun Microsystems, How to Implement a Provider for the Java™ Cryptography Architecture (http://java.sun.com/j2se/1.4/docs/guide/security/CryptoSpec.html). Last updated in May 2001.

[7] Knudsen, J., Java Cryptography. O'Reilly & Associates, first edition, May 1998.

[8] Callas, J., Donnerhacke, L., Finney, H., & Thayer, R., OpenPGP Message Format. Network Working Group, RFC 2440, November 1998.

[9] Attkins, D., Stallings, W., & Zimmermann, P., PGP Message Exchange Formats. Network Working Group, RFC 1991, August 1996.

[10] Zimmermann, P., An introduction to cryptography. Network Associates Inc., PGP, version 6.5.1 documentation, June 1999.

[11] McGraw, G., & Felten, E., Securing Java: Getting Down to Business with Mobile Code. John Wiley & Sons, second edition, January 1999.

[12] Mani, J, Shannon B., Spivak, M., Carter, K., & Cotton, C., JavaMail™ API Design Specification. Sun Microsystems, Inc., Version 1.2, September 2000.

[13] Rosen, K., Discrete Mathematics and Its Applications. McGraw Hill College Div, fifth edition, April 2003.

[14] Hall, M., More Servlets and JavaServer Pages. Pearson Higher Education, December 2001.

[15] Kurniawan, B., Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions. SAMS, April 2002.

[16] Oracle Corporation, the Oracle Technology Network web site (http://otn.oracle.com/index.html). Last updated in October 2003.

[17] Cryptix, the Cryptix web site (http://www.cyptix.org). Last updated in August 2001.

[18] International Business Machines Corporation, Microsoft Corporation, & VeriSign Inc., Web Services Security (WS-Security). Version 1.0, April 2002.

[19] Elmasri, R., Fundamentals of Database Systems. Addison Wesley, third edition, August 1999.