ENHANCED JAVA SECURITY TOOLS

By

IP-KIN ANTHONY WONG

Associate of Liberal Arts
Maui Community College
Kahului, Hawaii
1996

Bachelor of Science
University of Central Oklahoma
Edmond, Oklahoma
1998

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
December, 2001

ENHANCED JAVA SECURITY TOOLS

Thesis Approved:

[illegible]

Thesis Adviser

[illegible]

[illegible]

[illegible]

Dean of the Graduate College

PREFACE

This research talks about enhancements on the current Java security tools, key management, and random seeders. Its primary focus is on the Java security tools. The enhanced Java security tool suite will use the improved key management scheme and authentication technique discussed in this paper. In addition, the extended signing and verification functionality will be embedded inside this suite as well. Since the gist of this paper is about enhancements on the Java security tools, it is named Enhanced Java Security Tools (EJST).

This paper is organized into six chapters and three appendixes. Chapter 1, Introduction, depicts the background of Internet security, the current problems in Java security, and the objectives on this research. Chapter 2, Literature Review, introduces fundamental concepts and background knowledge on authentications, digital certificates and public key infrastructure (PKI), message digests and digital signatures, cryptographic algorithms, and the Java security model. Chapter 3, EJST, presents solutions to meet the objectives. It also describes the design and implementation of the EJST. Chapter 4, User's Menu, shows the user menu of the EJST's Key-Certificate-Policy manager (KCPM). Chapter 5, Conclusion, draws a conclusion on the EJST. Chapter 6, Future Work, talks about future improvements that can be done on the EJST. Appendix A, Acronyms, show the acronyms used in this paper. Appendix B, Glossary, provides some terminologies used in this research. Appendix C, Source Code, presents some of the source codes that are used on the implementation.

ACKNOWLEDGEMENTS

I wan to express my sincere gratitude to Dr. Mayfield, my principal adviser, for giving me invaluable advice, assistant, encouragement throughout my graduate study. His guidance and generous aid helped make this work possible.

I also want to express my appreciation to Dr. Lu, and Dr. Chandler who gave me support and advice to guide me through this thesis. They help me to organize my work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Background

Information security is an important piece in human life. Individuals can keep their own information secure by keeping their mouth shut. Organizations can store their files in a secure cabinet and only allow trusted employees to gain access to these files. Ever since the Internet emerged and was made available to the public, information security has become a huge issue. The Internet is an excellent vehicle to enhance information sharing, business-to-business transaction, and business-to-consumer transaction. However, the problem is that by connecting to the Internet, a door is opened wide for information hackers. The major potential victims are business firms and government agencies since they have information that could be worth millions of dollars. These attackers can hack into organizations through the Internet. They can eavesdrop the communication channels and disguise themselves as system users after they collected enough information. Once they successfully hack into the host system, they can cause different damage such as stealing corporate information and selling it to competitors, changing the data on the host system, or stealing customer information and using it for their own purposes.

Internet security has been a popular topic for years. A lot of research have been done and much more is underway. There are many commercial products available in the market. Some of these products include firewalls, which provides securities for communications at the application level as well as at the IP level; secure socket layer (SSL), which provides a secure channel for business transactions and prevent man in the middle attack; and encryption and digital signature, which provide authentication, data integrity, and data confidentiality.

Java has become one of the most popular programming languages. For the Internet, it provides support to all the three protocols mentioned above. In addition, Java provides tools and application programming interfaces (APIs) for SSL, message digest, encryption, digital signature, digital certificate, key management, and more. Java is designed to be secure. Toward this end, a lot of emphasis has been placed on security to provide virus-free and tamper-free systems [38]. Nonetheless, the main focus of Java security is to protect the information on a computer while still allowing Java program to run. The Java sandbox model was introduced to address these issues. The idea behind this model is that when a program is hosted on a computer, the host computer provides an environment where the program can be played (run), but it confines the program's play area within certain bounds [24]. Since Java's JDK 1.1, this security model had been expanded beyond the sandbox paradigm. Public-key and secret-key encryption, digital signatures, and digital certificates play important roles on this new model. This is because they provide authentication of who actually provided the Java class and data integrity of what originally intended to be sent. They provide end users and system administrators with the ability to grant specific privileges to individual classes or signers. They also offer users the ability to verify the integrity of classes. In addition, these classes can be used for other applications that require a serious authentication protocol. For example, instead of using simple passwords, a bank transaction system might require a more secure authentication protocol such as authentication using public-key encryption or authentication using digital signatures [37, 23, and 24]. By using public-key encryption and digital signatures, the bank can be sure that a transaction request came from the account holder, and it also can verify that the transaction has not been tampered with or altered.

Since the first release, Java has embedded a security model in its language. Java 1's JDK 1.0 default security model contains features that can prevent access from programs that may harm users' computing environment that may discover private information on the host computer [24]. Authentication was added to Java 1 JDK 1.1, and encryption was made available as an extension to Java 2 SDK 1.2.2. The latest version of Java is Java 2 SDK 1.3. It extends SDK 1.2.2's security model by providing more and improved security tools and security APIs.

1.2 Current Problems

The Java security model provides a framework for accessing and developing cryptographic functionality as well as managing security on the Java platform. It offers security tools and classes from the Java Security API for encryption, key generation, key management, digital signature, digital certificate, access control, and more.

Although the Java security model has been improved and provides many nice features, it is not airtight. A lot of security flaws, weaknesses, and limitations still remain. First of all, the Java security and cryptography packages reveal many loopholes and weaknesses on key management, authentication, and pseudo-random number generation. Second, the security tools provided by Java 2 are not user-friendly. They require the users to have extensive knowledge of the tools they are using. These tools also lack of a good help menu; thus they can be very confusing to use. Finally, Since the Java security model is designed primarily for Java applications and applets, the Java security tools can only sign and verify JAR files. This limits the usage of the Java security tools.

1.3 Objectives

Security loopholes on key management in addition to user interface unfriendliness and limitation on the Java security tools reveal many concerns. These problems not only barricade the usage of these tools, but also threaten the safety of the system. The purpose of this research is to address these problems.

The first objective is to provide a truly random number seeder so that patterns on pseudo-random numbers are less obvious. The second objective of this project is to provide a more secure key database key management scheme. This includes improvements on the authentication and key storage techniques. These enhancements will help provide the users with better security features. The third objective is to improve the Java security tools such as *keytool*, *jar*, *jarsigner*, and *policytool*, in terms of user interface friendliness, usability, and online help support. In addition, a better security tool management scheme will be used so that managing keystores, creating JAR files, signing and verifying files, and managing security policies can be done with great ease. The last objective is to extend message signing and data integrity verification to all file types, so that the Java security tools can benefit other applications as well.

2. LITERATURE REVIEW

2.1 PKI and Digital Certificate

In the banking industry, checks and certificates are handled through local banks and central banks. On the Internet, digital certificates are handled through some trusted organizations called certificate authorities (CA). It forms a network of trusted certificates and certificate chains.

Digital Certificate

Digital Certificate is a software token that carries information between applications. It offers a high level of authentication. It contains a user's credentials and public key to validate his identity [19].

X.509 Digital Certificate is the most popular standard for public key certificate. It was developed by the International Standards Organization (ISO) [9]. X.509 v3, the latest version of X.509 Digital Certificate, is a revision of the CCITT X.509 certificate standard. X.509 v3, as mentioned by Brieva, "extends the certificate with provisions that facilitate explicit management of certificates, certification paths, security policies, and the transfer of trust" [19]. A certificate includes the issuer name, the subject name, and the subject public key; and the certificate is signed with the issuer's private key. If, for example, party A has party B's certificate and knows the issuing CA's public key, he can verify party B's certificate and then use party B's public key to verify party B's signature on any document. X.509 v3 certificate can hold any number of extensions. Each extension has a criticality flag. If a certificate contains a critical extension, a certification path verifier that attempts to verify that certificate must be able to process that extension, or must not verify the certificate at all [19].

PKI

PKI is a set of standards and technologies for user authentication and secure methods of exchanging information [7]. Levitt said, "PKI encompasses a broad spectrum of technologies with dizzying array of possible applications" [5]. It allows businesses to use digital certificates to confirm identities. Levine explained that digital certificates are used to "ensure the confidentiality and integrity of data through encryption, control access through private keys, authenticate documents via digital signatures, and enforce nonrepudiation of business transactions." PKI secures sessions between the web browser and the web server. It controls who does what by issuing digital certificates. It relies on public-key cryptography to protect data that is sent electronically and digital certificates to validate user identities [6]. Yasin explained that, "Public keys can be distributed and used by a person to encrypt data such as e-mail. A receiver of a message uses his or her corresponding private key to decrypt data. By using a digital certificate electronically signed by a certificate authority, a user can authenticate himself or herself to another person or entity over the Internet" [7]. GTE CyberTrust PKI system, for example, requires a client to go through a three-tier process. First, a digital certificate is issued to an individual to establish a unique online identity. Second, two Ids are issued consisting of an individual password. Third, users are granted access only to specific functions [6].

There are two major PKI paradigms in the industry: hierarchical and horizontal. X.500/X.509 PKI is organized hierarchically. It is spanned like a tree with a Root Certificate Authority (RCA) serving as the root. The trust is centered at the root and is transferred hierarchically to all the users in the network via Certification Authorities (CA). The public key of the RCA is known to all the users, and it is used to induced confidence in the public keys of the other entities via some trusted paths in a trusted graph. X.500/X.509-based PKI is best suited for Business-to-Business (B2B) and Business-to-Consumer (B2C) environment.

PGP-based PKI, on the other hand, is organized horizontally. It does not specify any specific structure for a trusted graph. Users are free to decide whom they trust. PGP-based PKI uses a decentralized system of trusted introducers, which are analogous to CA in the X.500/X.509-based PKI. PGP-base PKI allows people to sign anyone else's public key. Unlike the X.500/X.509-base PKI, the PGP-based PKI is just a collection of all the keys in the user population, all the signatures on those keys, all the individual opinions of each PGP-based PKI user as to whom they choose as trusted introducers, all the PGP-base PKI client software, which runs the PGP trust model and performs trusted calculations for the client user, and all the key servers which disseminate this knowledge. No one will be fooled by a bogus key signed by an suspicious introducer in the PGP-based PKI model because one can tell if a key is certified by an introducer who he trust by looking at the introducer's signatures [32].

2.2 Message Digest Algorithm and Digital Signature

The most common way to establish proof of identity is through password-based authentication as discussed in the authentication section. A more secure avenue to establish proof of identify is through message digest and digital signature. Data confidentiality, data integrity, and nonrepudiation can be accomplished by using these two algorithms.

A one-way hash function, also called a message integrity check, a message digest function, a message digest algorithm, or a message digest, takes an arbitrary length plaintext as input and outputs a relatively small fixed-length string. This string is called a hash value or a cryptographic check, which serves as a unique fingerprint of the message. Each unique message fed to a one-way hash function is guaranteed to produce a unique hash value. Given this hash value, it is virtually impossible to generate the original plaintext; thus it is called a one-way hash function [28].

One-way hash functions can be used to provide a stronger authentication scheme. Instead of storing passwords, host computers can store the hash values of the passwords. When a user sends the host his or her password, the host computer performs a one-way hash function on the password. Then the host compares the result of the one-way hash function to the value it previously stored. Since the host computer only stores the hash value to the user passwords, the threat of someone breaking into the host computer and stealing the password list is reduced [37]. MD2, MD4, MD5, HAVAL, and RIPE-MD are example of the message digest algorithms.

MD5

The MD5 message digest algorithm (one-way hash function), developed by Ronal Rivest of RSA Data Security, Inc. (RSADSI) in 1991, is an updated version of MD4 [22]. A variable length message can be hashed to produce a fixed length, say 128-bit, message digest value. It is used to protect web servers with an RSA MD5 hash algorithm method [10]. The hash value calculation is optimized for 32-bit registers. Both MD4 and MD5 require padding to a multiple of 512 bits. The padding always includes a 64-bit value that indicates the length of the unpadded message [28]. It is primarily used to produce *fingerprints* of sets of data. Message digest authentication allows site manager to be more selective in their use of encryption and enable them to limit SSL session to data that truly needs to be protected. Unfortunately, it doesn't encrypt the traffic, and it merely hides passwords [10].

SHA

Secure Hash Algorithm (SHA), also called SHA-1, was developed by the U.S. National Institute of Standards and Technology (NIST). This algorithm is based on MD4. SHA produces a 160-bit message digest value, thus increases its protection ability [22].

Digital Signature

Digital signature is a special encrypted code attached to an electronic message. It lets the recipients know that the person sending the message really is who he claims to be. A digital signature binds a person's identity into an asymmetrically encrypted private key. This private key is issued to only one bearer and is used to digitally sign and encrypt a message. Someone with a valid public key can verify the identity of the message sender [8]. Digital signature works by utilizing a message digest algorithm, such as MD5, to calculate the message's hash value. The hash value is then signed by the sender's private key, and each digital signature is unique to the message it signs. Digital signatures can also provide integrity verification of a message because a signed message that has been altered will fail the recipient's signature verification [28]. Digital signature systems can be established within a PKI, and can be maintained by a certificate authority [8]. DSA, GOST, and ESIGN are examples of the public-key digital signature algorithms.

DSA

Digital Signature Algorithm (DSA) was developed by National Security Agency (NSA). It was released as a standard by the NIST. It is a combination of DSA and SHA-1 algorithms. Its key size varies from 512 to 1024 bits with a 64-bit increment [22].

2.3 Authentication

Authentication means establishing proof of identify. Usually, this involves one or a combination of items that a person knows, something that this person has, or something that this person is [28]. Traditionally, identities are established through passwords. Different password-based authentication protocols are discussed here. However, a more secure way is to use digital signature. This is discussed in the *Message Digest* and *Digital Signature* section.

Static Password in Cleartext

The most popular authentication technique employed on the Internet is based on static passwords. In this scheme, a user is given a user ID and an associated password. The host computer uses both to identify the users' identity. The users' IDs and passwords are sent to the host as a cleartext. Some host systems store their users' identity information in a file or in a database as cleartext. A more secure way is to store the password through a one-way hash algorithm. When a user enters his password during login, it is "crunched" through the one-way hash algorithm. If the result and the value stored in the password database are identical, the user must have entered the valid password. Unfortunately, this authentication scheme has several weaknesses: attackers who steal the password database can perform a dictionary attack to find a list of poorly chosen passwords. Passwords that are sent as a cleartext over a network, especially travel over the Internet, can be revealed right away after they have been eavesdropped [28].

Challenge-Response Static Password with One-Way Hash Algorithm

A challenge-response static passwords scheme is a much safer authentication protocol than the static passwords in cleartext authentication protocol. This is because a host can verify a user's identity without requiring him or her to send the password over the network. When a login request is received, the host issues a challenge string as a response. Upon receiving the challenge string, the user's client software concatenates the password he entered to the challenge string and computes a one-way hash, using MD5 for example, of the result. The output of the hash is then forward to the host, which independently performs the same calculation using the user's cleartext password. If the host's hash value matches the host's hash value, then the password he or she entered was correct. However, the problem is that the host needs to know the user's password in advance. In addition, the password container is still vulnerable to dictionary attack [28].

One-Time Password

Unlike static passwords, which are subject to network eavesdropping and dictionary attack, one-time password (OTP) is, in theory, immune to these attacks. There are three popular OTP mechanisms: Bellcore's S/Key, handheld authenticator, and smart card [28].

S/Key

S/Key OPT system is a software that was first conceived by Leslie Lamport, and later implemented by Phil Karn. It provides secure password-based authentication over insecure networks. It is generated by combining a seed with a secret password from the user and repeatedly applying a hash algorithm, such as MD5, to produce a sequence of passwords algorithmically, each of which can be used only one time [18]. An S/Key password can be calculated by the function $p = h(k)$ where k is the secret key, h is the hash function, and p is the result from the hash function. To allow a user to login n times, say n = 3, S/Key first stores this user's secret password. Then it applies the hash algorithm three times (n=3) to the secret password. The result h(h(h(k))) = h^3 (k) is stored in the S/Key password database. When this user logs in to the system the first time, he or she is prompted for his one-time password h2 (k), which is transmitted in cleartext. Upon receipt, the S/Key system hash the value once to calculate h^3 (k) = h(h^2 (k)). If this value matches the value that is stored in the password container, the user is authenticated. The next time this user logs in, he or she will be authenticated by supplying h(k). Unfortunately, the S/Key password database, which contains hashed secret password, is not without loopholes: a poorly chosen secret password is still subject to dictionary attack [28].

Handheld Authenticator

A handheld authenticator, also know as a handheld password or a token, is a small handheld device that generates OPTs. There are four types of the handheld authenticators and they all require that both the host and the authenticator know the common algorithm for calculating the OTPs in advance. These four authenticators are: asynchronous, synchronous, PIN (Personal Identification Number)/asynchronous, and PIN/synchronous. For the PIN/asynchronous scheme and the PIN/synchronous scheme, a PIN is required before generating a valid password, and the PIN is used to authenticate the user to the handheld authenticator but not the host. Handheld authenticators can disable themselves after many consecutive incorrect attempts in order to protect against the PINs they held [28].

Smart Card

A smart card is similar to a handheld authenticator in purpose. However, it is more sophisticated, more intelligent, and more expensive than a handheld authenticator. As described by Hughes, it contains "a CPU, miniature operating system, clock, some program ROM, scratchpad RAM for cryptographic calculations, and nonvolatile RAM or EEPROM (electronically erasable programmable read-only memory) for key storage" [28]. Smart cards compute OTPs in response to a challenge from a host. They communicate directly with the challenging entity through a smart card reader. After a user enters his or her PIN, the smart card reader processes the challenge and enables authentication to take place without further human intervention.

2.3 Cryptographic Algorithm

Cryptography is the science of enabling secure communication. A cryptographic algorithm, also called a cipher, is a mathematical function for encryption and decryption. A plaintext, also called a cleartext, is an unencrypted message. A ciphertext is an encrypted message. A cryptosystem is a cipher plus all possible plaintexts, ciphertexts, and keys. Some of the most popular cryptographic algorithms are discussed in this section.

Symmetric Algorithm

Symmetric algorithms, also called secret-key algorithms, conventional algorithms, single-key algorithms, or one-key algorithms, are algorithms where the encryption key can be calculated from the decryption key and vice versa [37]. A symmetric algorithm uses the same key for both encryption and decryption. A plaintext message is encrypted through a secret key and symmetric encryption algorithm to produce a ciphertext. To decrypt this ciphertext, the same key is used together with the corresponding symmetric decryption algorithm to generate the original plaintext. Note that the encryption and decryption algorithms must belong to the same cryptosystem. Before secure communication can be established, symmetric algorithms require that the sender and receiver agree on a secret key [28]. The security of these algorithms relies on the key; therefore, the key must remain secret. Encryption is denoted by $E_k (M) = C$ and decryption is denoted by $D_k (C) = M$ where M is the plaintext message, C is the ciphertext, E is the encryption algorithm, D is the decryption algorithm, and $_k$ is the secret key [37]. IDEA, RC2, RC4, DES, and Blowfish are examples of the symmetric algorithms.

Public-key Algorithm

Public-key algorithms, also called asymmetric algorithms, are designed so that encryption and decryption rely on different keys, and the decryption key cannot be calculated from the encryption key [37]. However, this is an overstatement; because given enough time and resources, a decryption key can be cracked in a feasible time period if its key size is not too large. The encryption key is called the public key and can be disclosed to the public. The decryption key is called the private key and must be kept in secret. However, sometimes a plaintext message can be encrypted by a private key and the resulting ciphertext can be decrypted by a public key. Digital signature is a typical example for this reverse role of the public and private keys. Encryption using public key $_{puk}$ is denoted by $E_{puk} (M) = C$ and decryption using public key $_{prk}$ is denoted by $D_{prk} (C) = M$ where M is the plaintext message, C is the ciphertext, E is the encryption algorithm, and D is the decryption algorithm. Diffie-Hellman, RSA, ElGAMAL, and Elliptic Curve are examples of the public-key algorithms.

Stream and Block Cipher

Symmetric ciphers can be divided into two categories: stream cipher and block cipher. A Stream cipher operates on one bit, byte, or word at a time, where as a block cipher acts on groups of them. A Block cipher has four operation modes: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB). ECB is the most basic and least secure mode, such that a given block of plaintext, any key from the keyspace would always encrypt to the same block of ciphertext. CBC is similar to ECB, but it encrypts blocks using the plaintext, the key, and a third input derived from a XOR operation between the ciphertext of the previous block and the plaintext the current block; thus ciphertext blocks are chained and patterns are hidden. CFB is similar to CBC; however, the plaintext is encrypted in segments that are smaller than the actual block size, and chaining occurs after encryption. OFB is similar to CFB; however, output from the preceding encryption is used instead of feeding back the preceding ciphertext [28].

DES

Digital Encryption Standard (DES) is a symmetric cipher developed by IBM. It is first published in 1975, and adopted by the U.S. government as the federal standard for the encryption of commercial and "sensitive-yet-unclassified" government data in 1977. DES is a block cipher and can be used in any of the four modes (EBC, CDC, CFB, and OFB). It is designed primarily for hardware implementation. It relies on a fixed-length 56-bit key that encrypts data in 64-bit blocks. The key consumes 64 bits as well, and it uses one bit in byte for parity, which is ignored by DES. Hughes [28] defined the DES algorithm in the following:

> *For encryption, a block of plaintext is first permuted, meaning that each bit swaps places with another bit. Then the 64-bit block is divided into left and right halves, or 32-bit subblocks. Next 16 rounds of calculations are applied to each half, with input from (unique per-round) 48-bit subkeys derived from the 56-bit key. Between rounds, the output from the left half becomes the input to the right half, and vice versa. After completing all rounds, the two subblocks are rejoined, and the result permuted to invert the initial permutation. A 64-bit ciphertext block emerges. Decryption is achieved through exactly the same sequence of steps, but with the order of the subkeys simply reversed.*

Unfortunately its weakness is its 56-bit key size from today's point of view. This makes it vulnerable to key search attacks [22].

Triple DES (TDES), also called DESede, is a more secure variation on the DES cipher. There are several variants on TDES. One variant uses two keys, doubling the effective key length to 112 bits; thus increase the keyspace by a factor of 2^56. Another variant uses three distinct 56-bit keys, tripling the key length to 168-bit, yielding an increase in the keyspace by a factor of 2^112 [28]. The three-key TDES encryption process, for instance, works as follows: first, encrypt a plaintext using the first key; second, decrypt the result of the first step by using the second key; third, encrypt the result of the second step by using the third key; finally, generate the ciphertext. For its decryption process: first, decrypt the ciphertext using the third key; second, encrypt the result of step one using the second key; third, decrypt the result of step two with the first key; finally, produce the original plaintext[22].

RSA

RSA, an acronym for the last names for its three creators: Ron Rivest, Adi Shamir, and Leonard Adleman, is a pubic key cipher. It is the first public key cryptosystem to offer both encryption and digital signature functionalities. RSA assumes that it is virtually impossible to factor the product of two very large numbers [28]. It mixes a number into a message, and then churns it in such a way that only the number's prime factors can undo the message. This number is then used to make a public key [19]. Public and private keys, or simply called key pair, are large numbers that are related mathematically and are generated by the RSA algorithm. Hughes [28] provided the following definition on the RSA algorithm:

> *To start, two large prime numbers, p and q, are selected and multiplied giving the product (or modulus) n. Next an encryption key e is chosen to be less than n and to have no common factors with the number (p-1)x(q-1). From e the decryption key d is then derived such that exd = 1x(mod(p-1)x(q-1))... The public key is the combination of the encryption key e and the modulus n, and the private key is d.*

In the public key cryptosystem if, for example, party A wants to receive a secure message from party B, he or she will create a key pair first and openly send the public key to party B. Party B then encrypts her message with the public key and send it back to party A. Finally, party A decrypts the ciphertext with the private key. RSA guarantees that only the private key can decrypt the ciphertext encrypted by the public key that are generated by the same key pair.

To make sure that the public key sent from the other end comes from the identity who claims who he or she is, one may require the public key be signed by a digital signature, which is another service that is provided by RSA. RSA uses a one-way hash function to generate a hash value, which can be encrypted in the sender's private key, producing a digital signature. The recipient can decrypt the signature using the sender's public key to reveal the sender's hash value, then calculate its own using the message that was received. If both, the sender's and the receiver's hash values are identical, the sender is authenticated since the hash value must have been encrypted in the sender's private key. It also guarantees that the message that arrives has not been undisturbed. Unfortunately, due to the complicated mathematical calculation, RSA runs a lot slower than symmetric ciphers like IDEA (around 1000 times slower). As a consequence, it is primarily used to complement symmetric cryptosystems when performing bulk encryption. For example, a sender can use DES cipher to encrypt a large plaintext message, with a randomly chosen key, and then use RSA cipher to encrypt that random key with the receiver's public key. After both, the key and the ciphertext, are received, the receiver can first decrypt the key using RSA, and then decrypt the ciphertext using DES [28].

RC2 and RC4

RC stands for either Ron's Code or Rivest's Code. RC2 is a block cipher and RC4 is a stream cipher. Both of them were developed by Ron Rivest. Both ciphers support variable-length keys. RC2 was designed to replace DES and it is two or three times faster than DES. It can operate in any of the four block modes and can perform triple encryption just like what TDES does. RC4 also runs faster than DES and was reverse-engineered sometime in 1994. Unfortunately, neither algorithm has gained popularity in the cryptographic community [28].

IDEA

International Data Encryption Algorithm (IDEA) was invented by Xuejia Lai and James Massey. It was originally named Improved Proposed Encryption Standard (IPES) in 1991. IDEA is a 64-bit block cipher with a 128-bit key. It can operate on all of the four block cipher modes. Hughes [28] has the following definition on IDEA:

> *IDEA encryption begins by dividing a 64-bit plaintext block into four 16-bit subblocks. Each subblock is subjected to a number of computational rounds, involving 52 different subkeys derived from the 128-bit key. There are eight rounds. The calculations in each are fairly simple, limited to XOR, modular addition, and modular multiplication... Between rounds, the second and third subblocks swap positions. After the final round, the four subblocks are*

concatenated to produce a 64-bit block of ciphertext. Decryption involves exactly the same steps in the same order, but uses subkeys that are derived differently.

DH

Diffie-Hellman (DH), developed by Whitfield Diffie and Martin Hellman, and it was published in 1976. It is the first algorithm to introduce in the public key cryptography. DH solves the key management problem, which is suffered inherently from symmetric key ciphers such as DES, IDEA, and RC4. It allows both parties, the receiver and the sender, to derive a key independently without exchanging any secret information [28]. Thus, snoopers will not know the value of the secret key, even if they are able to listen to the entire transmission between the two parties [22]. The process begins with, for example, party A and party B making consent on two large numbers (150 digits or more) with mathematical properties relative to each other. Then party A and party B independently select their own large random numbers that they keep. Next, both independently enter their own secret numbers, along with the two-shared numbers, to a function involving modular exponentiation. Both parties openly exchange their results, and each performs a second similar calculation with the each other's numbers. The results are the public keys, which are identical for both parties [28].

2.4 Java Security Model

2.4.1 Overview

The Java security model was designed to offer three major security measures. The first one is the language design features such as strong type conversion, array bound checking, pointer arithmetic elimination, and strong memory protection. The second feature offers a sandbox mechanism to control program access. The final feature offers encryption and digital signatures for code owners or administrators to attach their certificates to Java classes; thus providing the end users and the hosts authentication and data integrity [23]. The Java security is comprised of the following:

1. The class file verifier

   The class file verifier ensures proper formatting of the program code. It contains a bytecode verifier, which verifies that the bytecode does not violate the type safety restrictions of the Java Virtual Machine (JVM), that the internal stacks cannot overflow or underflow, and that the bytecode instructions will have correctly typed parameters [23].

2. Class loader

   The class loader decides when, where, and how the codes can be loaded by the Java program. It ensures that system-level components within the run-time environment are not replaced [23]. It is responsible for loading classes that are found on the CLASSPATH as well as the classes that cannot be found on the CLASSPATH [24].

3. Security manager

   The security manager is the primary interface between the Java core API and the operating system. It has the responsibility for preventing or allowing run-time access to all system resources [24]. As listed by Postoia [23], these resources include "file I/O, network I/O, create a new class loader, manipulate threads and thread groups, start processes on the underlying operating system, terminate the JVM, load non-java libraries (native code) into the JVM, perform certain types of windowing system operations and load certain types of classes into the JVM." Starting from Java 2, policies to be enforced by the security manage can be specified in a file called *java.policy*; thus determining security policies becomes more flexible.

4. The access controller

The access controller enforces the security policies base on the entries on the security policy file. It also provides a much simpler method of granting "fine-grained" and specific permission to particular classes [24]. The access controller gives a simple procedure for giving specific permissions to specific code. To enforce security, the Java API calls the methods of the security manager; but behind the scene, most of these methods call the access controller [23].

5. The security and cryptography packages

The security and cryptography packages form the basis for authenticating and integrity checking for signed Java classes. It is composed of a collection of general-purpose classes for cryptographic functions collectively known as the Java Cryptographic Architecture (JCA), a collection of advance classes for advance cryptographic functions collectively known as the Java Cryptographic Extension (JCE) [23], and a collection of classes for access control. These three groups of classes provide APIs to form the Java Security API. The Java Security API is the gist of this paper. More details about the Java Security API will be discussed later in this section.

6. The key database

The key database is actually part of the security and cryptography packages. It used to store key entries and certificate entries. A key entry contains a private key and the associated certificate chain. A certificate entry contains a single trusted certificate. These key and certificate entries can be used by the security manager and access controller to verify the digital signature that accompanies a signed class [24]. Java 2 provides an API for the key database, called *keystore*; thus key database and keystore are referring to the same thing. Note that this is an engine class. Java 2 also ships a concrete implementation of the keystore. It has the type *JKS* and the provider *SUN*. Figure 2-1 shows the anatomy of a typical Java application. Note that the class file verifier, the class loader, the security manager, and the access controller are parts of the Java Run-time Environment (JRE).

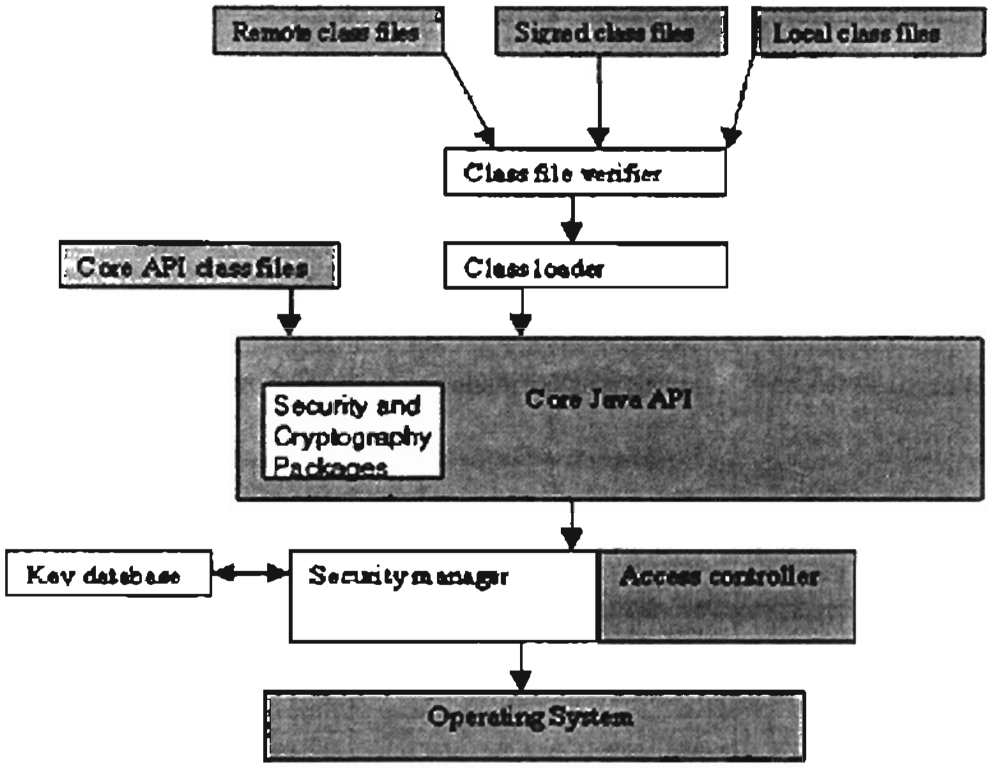
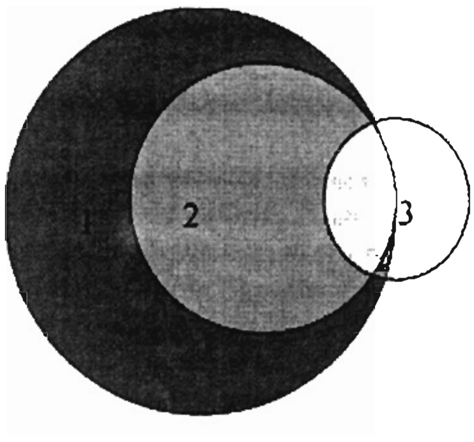


Figure 2-1. Anatomy of a typical Java application

The set of core classes in Java 2 can be divided into the set of security-related core classes and the set of other core classes. The security-related core classes can be further subdivided as access control and permission related core classes and cryptography-related core classes. The Java Security API is comprised with the classes from the union of circle 1 and circle 2 as displayed in figure 2-2. It includes all the classes related to access control and permission and all the classes related to general and advance cryptographic classes. JCA is a subset of the Java security model. It contains only the general cryptographic classes. JCE is an extension of JCA. It contains the rest of the cryptography-related classes, which are more advanced and are subject to export restrictions [23]. Figure 2-2 shows the relationship between the APIs in Java 2.



1. Java 2 SDK APIs
2. Java 2 SDK security-related APIs
3. Java 2 Cryptography-related APIs

Figure 2-2. Relationship between the Java 2 SDK, JCA, and JCE APIs

2.4.2 JCA, JCE, and Access Control

Java Cryptography Architecture (JCE) is comprised with a set of general-purpose cryptography-related classes to perform operations such as key generation, message digest calculation, and digital signature creation. These classes provide public methods as interfaces for programmers known as APIs. JCA is a structure for accessing and developing functions on cryptography for the Java platform. It was designed to provide implementation independence and interoperability and algorithm independence and extensibility. Implementation independence is achieved by using the "provider-based architecture." As described by Postoia, "the term cryptographic service provider (provider for short) refers to a package or a set of packages that supply a concrete implementation of a subset of the cryptography aspects of the Java security API" [23]. These packages must implement at least one cryptography service such as public-key algorithms and digital signature algorithms. Implementation interoperability means, as defined by Postoia, "various implementations can work with each other, use each other's keys, or verify each other's signature" [23]. Base on this definition, digital signature generated by one provider can be verified by another, and a key generated by one provider can be usable by another using the same algorithm. Also, Postoia [23] explained that:

> *Algorithm independence is achieved by defining types of cryptographic services, and defining classes that provide the functionality of these cryptographic services. These classes are called engine classes, and examples are the MessageDigest, Signature, and KeyFactory classes.*
>
> *Algorithm extensibility means that new algorithms that fit in one of the supported engine classes can easily be added.*

Java Cryptography Extension (JCE) is an extension of the JCA. It is comprised with a set of advance cryptography-related classes such as encryption, key exchange, and message authentication code (MAC). JCE relies on the same architecture as JCA does. This means that it also offers implementation independence, implementation interoperability, algorithm independence, and algorithm extensibility.

The Java Security API contains classes that only concern with access control, security policy, and permissions. Although they are not related to cryptography, they play a major role in the Java Security Model.

2.4.3 Java Security Tools

Java 2 provides four security tools: *keytool*, *jar*, *jarsigner*, and *policytool*.

The *keytool* tool is a command line utility. It provides an administrative interface to manage keys and certificates in a keystore. Through the *keytool* administrative interface, users can perform operations on the keysotre. These operations include: create key pairs and self-signed certificates, export certificates to be sent to others along with the signed message, issue certificate signing requests (CSRs) to be sent to CAs for signing, import certificates to verify signatures, install certificate chains from certificate replies, create new keystores, change passwords to the keystores and the key pairs, and remove key pairs and certificates [23].

The *jar* tool is another command line utility. It is used to compress and pack files into JAR files. In addition, jar can also be used to extract files from JAR files. The compression is done based on ZIP and ZLIB compression format. During the archive process, *jar* can create a special text file, called JAR manifest or simply manifest, which contains descriptions of each file archived in the JAR file [23].

The *jarsigner* tool is another command line utility. It is used to sign JAR files and to verify signatures and the integrity of signed JAR files [23]. Behind the scene, jarsigner uses the entries in a keystore to look up information about a particular identity and uses that information either to sign or to verify a JAR file [24].

The *policytool* is a GUI-based utility. It is used to create and manage security policies, which are stored in a policy file. These security policies are used to grant permission to various Java codes (i.e. class files and JAR files) depending upon the code base and/or the digital signature applied to the code [23]. For more details, see User's Menu in a later chapter.

2.4.4 Pseudo-random Number Generator

Java 2 contains two pseudo-random number generator (PRNG) classes: Random and SecureRandom. A Random object can be initialized with a seed that represents the starting point for the random number sequence. A program that uses the same seed value gets the same sequence of numbers from the generator. In addition, if no seed value is provided, the Random object uses the value of the system clock as the seed value. As

Knudsen pointed out, "This is a predictable seed... If an attacker knows when you create the random number, even approximately, he or she can guess at likely values of the random number seed" [22]. A SecureRandom object is cryptographically strong; thus it is more secure. The SecureRandom object uses a digest algorithm to digest the seed valuc. Then, the resulting digested value is stored as part of the SecureRandom object's internal state. When a random number is requested, the SecureRandom object updates the message digest with the internal state and the internal counter. The SecureRandom class contains a seed generator, which is used to generate seed values for new SecureRandom objects. The seed generator uses the timing of the thread to seed itself [22].

3. ENHANCED JAVA SECURITY TOOLS (EJST)

Recall from the Introduction, this paper has four objectives. First, provide a truly random seeder. Second, provide a more secure key management scheme through stronger authentication techniques and a better key storage algorithm. Third, provide enhancements on the existing Java security tools. Fourth, extend message signing and data integrity verification to all file types.

3.1 Problem and Solution Overview

3.1.1 Pseudo-Random Numbers

Currently, Java 2 offers both the standard pseudo-random number generator (PRNG), which is represented by the Random class, and a more secure version of the standard PRNG called SecureRandom, which is represented by the SecureRandom class.

Numbers that are generated by the Random class can be compromised easily because the seed value is not generated at random as discussed earlier. Numbers that are generated by the SecureRandom class are more secure because seed values are generated quite randomly. Unfortunately, the SecureRandom class has not been tested thoroughly. As Knudsen pointed out, "It may have weaknesses that cryptanalysts could exploit" [22]. In addition, both the Random class and SecureRandom class calculate their seed values through algorithms. Since most computers are deterministic machines, seed values that are generated from these machines are not truly random. As a consequence, random numbers generated from these pseudo-random number generators may reveal patterns. Attackers can utilize these patterns to compromise security systems that use these generators. On the other hand, human actions can be truly random; therefore, truly random seed values can be generated through human actions.

Human actions can be recorded through input devices such as: keyboard, mouse, scanner, and microphone. Here, only the keyboard and the mouse are utilized.

Human actions can be tracked through keyboard events such as the timing between successive keystrokes and the keystroke that got struck. Using the internal representation of the keys that got struck as the source is not very secure because it can be compromised if someone is watching behind the user's shoulder. The timing between successive keystrokes is a better choice because even a very consistent typist will probably not be able to type with millisecond precision. This means that the seed values generated from this kind of seeder can be truly random. This technique has been used for years in Pretty Good Privacy (PGP) [22].

Human actions also can be tracked through mouse events such as mouse movements and the timing between successive mouse clicks. The timing between successive mouse clicks is not very a wise choice because there are only a few buttons on the mouse. On the other hand, mouse movements are more random. This is because it is probably impossible to repeat exactly where the mouse has traveled. This means that the seed value generated from this kind of seeder can be truly random.

In addition to the two human action seeders mentioned above, truly random seeds can be generated from randomly chosen bits from a file. These three seeders are implemented into a suite of seeders, called the Truly Random Seeders (TRS).

3.1.2 Key Management

Java's key database, also called keystore, is used to store key entries and certificate entries. The private key in the key entry should be kept secret because compromising the private key will compromise the cipher text generated by that public-key algorithm. Unfortunately this keystore implementation provided by Sun Microsystems has a poor key management. One of the loopholes is the weak authentication scheme it uses. Currently, access to the keystore and the key entries require passwords. However, there is no restriction on the passwords used. As a result, the keystore and its key entries are subject to dictionary attack. A password guesser such as *crack* in Unix can be used to perform dictionary attacks.

Key management is the hardest part in cryptography. Cryptanalysts often attack both public-key and secret-key cryptosystems through key management [37]. Even the best key management system is vulnerable if the passwords to the key database or key entries are badly chosen. Here, we have two approaches to avoid badly chosen passwords. They provide stronger authentications to the keystore key management.

One can mandate that users enter passwords containing punctuation or numeric characters that are found in the American Standard Code for Information Interchange (ASCII) character sets. This approach works because a typical dictionary attack uses keys from a file, which contains words that can be found in a dictionary, to run against the stolen password file. Dictionary attack would not work if the passwords contain punctuation or numeric characters.

Alternatively, instead of asking the user to provide a password for each key entry, Schneier [37] suggested a key management scheme called *random keys*. A random key is created using a pseudo-random number generator, which generates a random string [37].

Using this technique, random passwords can be generated the same way random keys are generated. The Truly Random Seeders (TRS) and the SecureRandom random number generator can be used to generate random passwords. All passwords will be created as an eight-character long string as recommended by Knudsen [22]. Table 3-1 gives the number of possible keys with different constraints on the input strings, and table 3-2 gives the time required for an exhaustive search through all of the keys given a million attempts per second. Here the printable character sets, without the space character, will be used to represent the random passwords. An eight-character long string contains 94^8 or approximately $6.1*10^{15}$ possibilities. This not only is sufficient to defend against a dictionary attack, but also good enough to protect against an exhaustive key search attack, also called brute force attack. However, random keys with shorter key space are vulnerable, and random keys with long key space are hard to remember; thus eight-character long passwords should be adequate. The Key-Certificate-Policy Manager (KCPM), which will be discussed later in this chapter, utilizes the random password authentication technique to provide better authentication to the keystore and key entries.

Key storage is another problem in the keystore key management scheme. Currently, the keystore implementation provided by Sun Microsystems uses an internal algorithm to encrypt private keys. It uses the *KeyProtector* class to protect the private key in a key entry. This *KeyProtector* classes uses the password of the key entry and concatenates it with a *salt*. After a series of hashing, XOR, and concatenation operations, the key is claimed to be protected. Unfortunately, the salt is generated randomly and cannot be derived from the password. This salt is stored in the keystore file, together with the key, unprotected. If an attacker knows where the salt is stored, the private key can be easily compromised.

In fact, as Oaks [24] pointed out, "The strength of this encryption is limited; because it is part of the standard Sun distribution... 'Weak' is a relative term in this context; it still require some effort for the encryption to be broken, but it can be done." Furthermore, as Knudsen [22] explained, a private key is encrypted by scrambling the "passphrase", which is used as the password to access the private key, and combining it with the private key.

This research uses a more secure password-based encryption algorithm, called *PBEWithMD5AndDES*. This algorithm is implemented by Sun Microsystems, and it comes with the Java Cryptography Extension (JCE). The PBEWithMD5AndDES algorithm provides password-based encryption and decryption base on the RSA Laboratories PKCS #5 v1.5: Password-Based Cryptography Standard. This algorithm uses DES as the symmetric cipher, cipher block chaining (CBC) mode as the cipher mode, PKCSPadding as the padding scheme, and MD5 as the hash function.

For encryption, the PBEWithMD5AndDES algorithm uses the password of the private key to generate a secret key. Then this secrete key is used with the DES symmetric cipher to encrypt the private key before it is saved to the keystore. Decryption is done in the same fashion. Through the PBEWithMD5AndDES algorithm, private keys can be securely protected. This feature is implemented in the Secure Keystore, which will be discussed later in this chapter. The Secure Keystore is an alternation of Sun's keystore implementation. It embeds the PBEWithMD5AndDES password-based algorithm to protect the private keys.

3.1.3 Java Security Tools

Java 2 provides four security tools: *keytool*, *jar*, *jarsigner*, and *policytool*. The *keytool* is a command line utility. It is used for keystore management. The *jar* tool is another command line utility. It is used to archive Java class files. The *jarsigner* tool is also a command line utility. It is used to sign and verify JAR files. The *policytool* is a GUI-based utility. It is used to grant permissions to Java applications and applets.

Although *keytool*, *jar*, and *jarsigner* provide many different kinds of features, their user interfaces are not user-friendly. Many of their operations have long commands. For example, generating a key pair in *keytool* may require up to 22 parameters. Users need to remember the spelling of the keyword and the order of the arguments. These inconveniences make the command line utilities, mentioned above, difficult to use. In addition, all of the four security tools lack a good help menu. The help menus provided by these tools are very brief. The content on these help menus only provide descriptions on the syntax of the tools. They do not explain what the operations are, what they do, and what they are used for. Furthermore, the Java security tools, namely *jarsigner*, can sign and verify JAR files only. This limits the usage of the Java security tools.

This research provides graphical user interfaces to replace the command line user interfaces found in *keytool*, *jar*, *jarsigner*. These GUIs are user-friendlier. They can eliminate problems that are encountered on complex operations. They can also enhance the usability on these tools. In addition, this research extends message signing and data integrity verification to all file types, so that the Java security tools can benefit other applications as well.

Furthermore, this research provides a GUI-based on line help menu. This help menu is designed as web pages on a web browser. Users can browse the menu through links. This help menu also provides detail descriptions about each operation. This aid the users to get a better understanding of each process.

The GUI-based security tools are organized into an application. When this application runs, users can perform all the operations that can be found in the old security tools. This application is like a console, and it is called the Key-Certificate-Policy Manager (KCPM). The KCPM manages all the operations provided by the security tools and the underlying keystore as a whole; thus it provides easier user access and better management to the keystore as well as the security tools.

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	460,000	1.2 E7	3.1 E8	8.0 E9	2.1 E11
Lowercase letters and digits (36):	1,700,00 0	6.0 E7	2.2 E9	7.8 E10	2.8 E12
Alphanumeric characters (62):	1.5 E7	9.2 E8	5.7 E10	3.5 E12	2.2 E14
Printable characters (95):	8.1 E7	7.7 E9	7.4 E11	7.0 E13	6.6 E15
ASCII characters (128):	2.7 E8	3.4 E10	4.4 E12	5.6 E14	7.2 E16
8-bit ASCII characters (256):	4.3 E9	1.1 E12	2.8 E14	7.2 E16	1.8 E19

Table 3-1. Number of possible keys of various keyspaces

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	.5 s	12 s	5 m	2.2 h	2.4 d
Lowercase letters and digits (36):	1.7 s	1 m	36 m	22 h	33 d
Alphanumeric characters (62):	15 s	15 m	16 h	41 d	6.9 y
Printable characters (95):	1.4 m	2.1 h	8.5 d	2.2 y	210 y
ASCII characters (128):	4.5 m	9.5 h	51 d	18 y	2300 y
8-bit ASCII characters (256):	1.2 h	13 d	8.9 y	2300 y	580,000 y

Table 3-2. Exhaustive search of various keyspaces with one million attempts per second

3.2 EJST Architecture, Design, Implementation, Installation

The Enhanced Java Security Tools (EJST) is a tool suite. It provides three utilities: Truly Random Seeder (TRS), Secure Keystore, and Key-Certificate-Policy Manager (KCPM). The TRS provides truly random seeds to any pseudo-random number generator to reduce patterns. The Secure Keystore utilizes the PBEWithMD5AndDES password-based encryption algorithm to provide a more secure key storage. KCPM is designed to replace the user interfaces found in the command line security tools. It provides GUIs that are user-friendly. It contains an online help menu to aid the users. It embeds the password restriction and random password techniques to provide stronger authentication for the underlying keystore. Furthermore, It extends data integrity checking to all file types.

3.2.1 Truly Random Seeders (TRS)

The Truly Random Seeders (TRS) is used to generate some truly random seed values. These values can be used to seed the pseudo-random number generators. Since the seed produced by the TRS can be truly random, random numbers that are generated by the pseudo-random number generators that use the seeds from TRS may produce less obvious patterns or at least makes these numbers less predictable [22].

The Truly Random Seeder is a suite of random number seeders. It contains the Keyboard Seeder, the Mouse Seeder, and the File Seeder.

The Keyboard Seeder utilizes the keyboard timing. It calculates the seed by measuring the time between successive keystores using a fast timer with a resolution of one millisecond. The *currentTimeMillis* method from the *lang.System* class has a resolution of 10 milliseconds. This is insufficient. As a result, the *Counter* class is implemented to provide the needed resolution. It creates a thread for itself and increments the counter data member once every millisecond.

The Keyboard Seeder has a byte array data member. It is used to represent the value of the seed. The length of this byte array is defined the same as the length of the seed. The timing between keystrokes is assigned to the next unassigned byte on the byte array. The Keyboarder Seeder rejects repeating keys because the timing of repeating keys may be predictable [22].

To use the Keyboard Seeder, the user needs to create a *KeyboardSeeder* object and specify the number of bytes he wants. For example, to generate an eight byte seed, the user need to hit the keystrokes on the keyboard, without consecutively repeating keys, eight times. Then, by calling the *getSeed* method, the seed is returned.

The Mouse Seeder utilizes mouse movements. It calculates the seed by measuring successive movements of the mouse pointer. The location of a mouse pointer is represented by a point, which contains an X coordinate and a Y coordinate in pixels.

The Mouse Seeder has a byte array data member. It is used to represent the seed. The length of this byte array is defined the same as the length of the seed. The X and Y coordinates are multiplied to produce a random number. This number is then assigned to the next unassigned byte on the byte array. It rejects repeating coordinates because these values may not be random.

To use the Mouse Seeder, the user needs to create a *MouseSeeder* object and specify the number of bytes he wants. Then the user is required to move the mouse pointer around on the screen. Finally, by calling the *getSeed* method, the seed is returned.

The File Seeder utilizes the bits in a file. It calculates the seed by randomly choosing the bits from a portion of a file. The File Seeder has a byte array data member. It is used to represent the value of the seed. The length of this byte array is defined the same as the length of the seed.

The File Seeder allows the users to choose any source file the user computer. If the file contains fewer bits than what is needed, an error message will be prompted. After the source file has been specified, File Seeder uses the *SecureRandom* to randomly choose the bits on the source file. These bits are then assigned to the byte array.

To use the File Seeder, the user needs to create a *FileSeeder* object, and specify the number of byte he or she wants. The user is then prompted to specify the source file. Finally, by calling the *getSeed* method, the seed is returned.

3.2.2 Secure Keystore

The Secure Keystore uses the PBEWithMD5AndDES password-based algorithm to provide more secure key storage to the existing keystore key management scheme [22]. The Secure Keystore provides services such as store and retrieve key entries, store and retrieve certificate entries, delete key and certificate entries, change password of the keystore and the key entries, check entry types, and get the number of entries stored in the keystore.

The Secure Keystore uses the PBEWithMD5AndDES password-based algorithm and the *KeyProtector* class to protect the private keys. Note that the *KeyProtector* class is used here only for backward-compatibility. When the private keys are stored, they are PBEWithMD5AndDES encrypted using the passwords to these keys as secret keys.

When the private keys are accessed, they are PBEWithMD5AndDES decrypted using the password to these keys as secrete keys.

Besides using private key encryption, the Secure Keystore uses SHA message digest algorithm to protect its integrity. When the keystore is loaded, the integrity is checked, and when the keystore is stored, the digest value is calculated and stored with the keystore.

The Secure Keystore needs to be installed before it can be used. First, the provider for the Secure Keystore is created. Its name is *EJSTProvider*, and it contains a keystore type *EJST*. Then the *EJSTProvider* is added to the *java.security* file, located at the ***java.home*\lib\security** directory, as a new security provider entry (***java.home*** is the location where Java is installed.) After these two steps, the Java Security API will know where to find the concrete implementation of the Secure Keystore. Note that the Secure Keystore is represented by the *Ekeystore* class

3.2.3 Key-Certificate-Policy Manager

The Key-Certificate-Policy Manger (KCPM) is an application console. It is designed to replace the user interfaces provided by *keytool*, *jar*, and *jarsigner*. It also provides easy access to the *policytool* utility. Through the KCPM, users can perform all the security operations in one application. Furthermore, KCPM provides an online help menu to aid users.

The KCPM provides a number of services. These services include generate key pair, delete key pair, change key pair password, delete certificate, import single certificate, import certificate chain, export certificate, certificate signing request (CSR), print certificate, sign JAR file, verify JAR file, sign regular file, verify regular file, create JAR file, security policy, create keystore, change keystore, change keystore password, list keystore entries, options, and help menu.

Each of these services is accessed by a dialog box. When a menu on the menu bar is clicked, an action event that is listening to that menu is fired, and the corresponding dialog box is displayed. A dialog box may contain many components such as text fields, labels, buttons, combo boxes, and radio buttons. Each component may be responsible for specific events. An event listener of a specific type is added to the component if the component is responsible for that event. Each of the listeners is represented by an inner class. Since the KCPM user interface listens to over one hundred events, there are over a hundreds inner class files crated in the directory where KCPM is located.

To provide stronger authentication to the underlying keystore, KCPM utilizes the password restriction and random password techniques. The users can set the password restriction on to enforce password restrictions. In addition, users also can request a random password from KCPM. The random password technique utilizes the TRS to get random seed values. These values are used to seed the SecureRandom object to produce strings of printable ASCII characters. KCPM uses these strings as random passwords.

The KCPM not only contains classes that handle operations on the graphical user interfaces, but also contains classes that perform the actual operations. The *Ekeytool* class is responsible for all tasks related to key management as well as signing and verifying regular files. It contains an instance of the Secure Keystore (*Ekeystore*) class, which is used for the underlying keystore management. The *EjarSigner* class is responsible for signing and verifying JAR files. The *Policytool* class is responsible for generate the *policytool* interfaces. The *RandomKey* class is responsible for generating random passwords and enforcing password restrictions. The *MouseSeeder* is responsible for generating random seeds.

3.4 <u>Installation</u>

EJST is a multi-platform application. It can be installed on any computer that supports Java 2. With a few configurations, EJST can be up and running. The followings are the installation steps on a Windows platform:

1. Make sure Java SDK 1.2.2 or later version and Java JCE 1.2.1 or later version are installed on the computer.

2. Copy the EJST directory, which contains all the application class files, to a directory on the user hard drive (i.e. C:\Anthony)

3. Add ***SET CLASSPATH=C:\JDK1.3\LIB;.;C:\ANTHONY*** to an empty line in the autoexec.bat file (assuming the operating system is running Windows 98, Java SDK is installed on C:\JDK1.3, and the EJST directory is installed on C:\Anthony)

4. Add ***security.provider.1=EJST.EJSTProvider*** to the provider section in the *java.policy* file located in the *java.home\lib\security* directory. If the security provider number is assigned, choose the next available number.

5. After the computer is rebooted, EJST is ready for use.

4 KEY-CERTIFICATE-POLICY MANAGER (KCPM) USER'S MENU

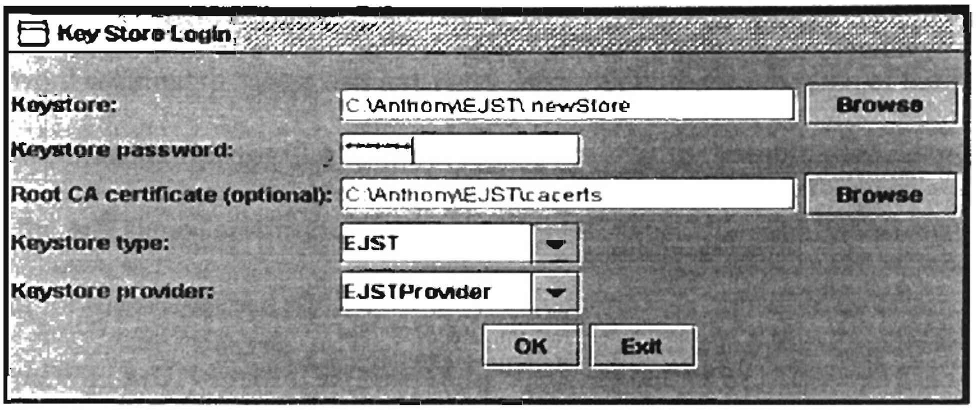
4.1 Overview

The Key-Certificate-Policy Manager (KCPM) is a GUI-based application. It is used to manage the keystore and the security tools. Users can utilize it to perform operations on public/private key pairs, digital certificates, JAR files, key stores, and security policies. It also provides the users with system options and online help menu.

When KCPM starts up, the **Keystore Login** dialog box, as shown it Figure 4-1, would pops up and looks for a specific user keystore file, root certificate authority (CA) keystore file, and the password to login to the user keystore. When KCPM is run at the first time, **Keystore** and the **Root CA certificate** are blank. Users can utilize the Browse buttons on the dialog box to search these two files. KCPM provides an empty user keystore, which is named *.store*. The password to this keystore is *keystore*. Users are recommended to change the password of the *.store* keystore. KCPM also provides a root CA certificate keystore, which is named *cacerts*. It contains five trusted root CA certificates from VeriSign. Figure 4-27 shows these ten VeriSign root CA certificates. By default, these two keystore files are stored in the same directory, where EJST class files located. **Keystore type** specifies the type of the keystore implementation. KCPM uses *EJST* keystore implementation, which is as the name suggests, created by EJST. Users also can choose *JKS* keystore implementation, which is, provided by SUN Microsystems, to work on user keystores that are compatible with the JKS implementation. **Keystore**

provider specifies the keystore implementation service provider (creator). The provider for EJST is *EJSTProvider*, and the provider for JKS is *SUN* [22, 23, 24].

After the user logs on to the specified user keystore, the KCPM's menu bar is enabled.



Keystore: C:\Anthony\EJST\ newStore
Keystore password: ******
Root CA certificate (optional): C:\Anthony\EJST\cacerts
Keystore type: EJST
Keystore provider: EJSTProvider

Figure 4-1. Keystore Login dialog box

4.2 Public / Private Key Pair

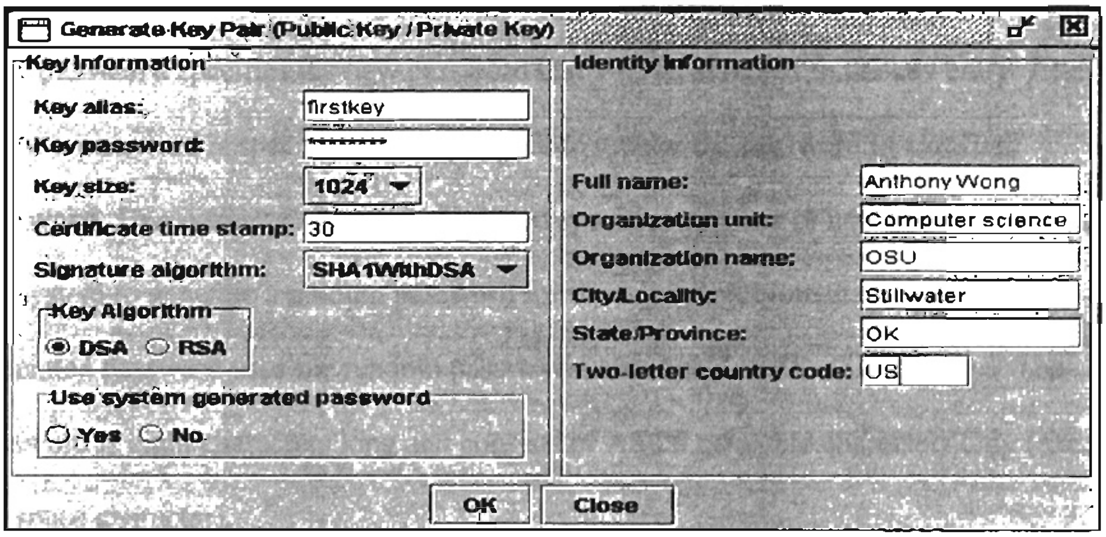
The **Public/Private key** menus provide services on generating public / private key pairs (or simply called key pairs), deleting key pairs, and changing the password to the key pairs.

Generate Key Pairs

The **Generate Key Pair** dialog box, as shown in Figure 4-2, creates a pair of public key and private key. The public key in the key pair is wrapped into an X.509 v1 self-signed certificate. This certificate is contained in a certificate chain, which has only one element: the self-signed certificate. To obtain a trust certificate chain from the CA, one should send a certificate signature request (CSR) to a CA and import the certificate chain from the certificate reply to the user keystore. The private key and the self-signed certificate is stored together in the user keystore as a key entry identified by an alias name. Note that generating a new key pair may take a few seconds due to the process of creating the public and private key pair. The bigger the key size specified, the longer the process takes [23, 24].

The **Key Information** section specifies the information about the key pair and the self-signed certificate. **Key alias** specifies the alias name, which is used to identify the entries (key entries and certificate entries) stored in a keystore. **Key password** specifies the password required to access this key entry. **Key size** specifies the size of the key pair. **Certificate time stamp** specifies the time stamp, which is the validity of the certificate in days, of the self-signed certificate. **Signature algorithm** specifies the digital signature algorithm for the self-signed certificate. KCPM supports four signature algorithms: *SHA1withDSA* , *MD2withRSA*, *MD5withRSA*, and *SHA1withRSA*. Although Java 2 comes with SHA1, MD2, and MD5 message digest algorithm and DSA public key cipher, it doesn't come with RSA public key cipher. As a result, users, who want to use the RSA cipher or RSA related signature algorithm should purchase the RSA cipher from RSA, Inc. and install it into their system. **Key algorithm** specifies the algorithm for generating the public key and the private key for the key pair. **Use system generated password** specifies whether the user wants to the system to generate a random password for the key entry or not.

The **Identity Information** section contains information about the identity of the owner (subject) of the key pair. Since the public key is wrapped around into a self-signed certificate, this information is used to specify the issuer of the self-signed certificate as well. Therefore, the owner of the key pair and the issuer of the self-signed certificate both refer to the same entity. To obtain a certificate issued by a CA one should create a certificate signature request (CSR). For details about the CSR, please refer to the CSR section later in the user menu. **Full name** specifies the common name of the individual, which is usually the first and last name of an entity. **Organization name** specifies the name of the organization with which the individual is associated. **Organization unit** specifies the unit with which the individual is associated. **City/Locality**, **State/Province**, and **Two-letter country code** specifies the city or locality, state or province, and country where the identity resides respectively. This information is used to create a X.509 distinguished name (DN) of a certificate [23, 24].

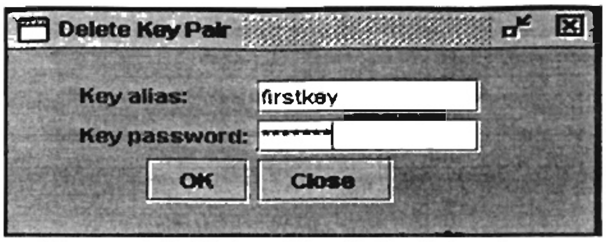

Generate Key Pair (Public Key / Private Key)
Key Information
Identity Information
Key Algorithm
Use system generated password
OK
Close

Key alias: firstkey
Key password: ********
Key size: 1024
Certificate time stamp: 30
Signature algorithm: SHA1WithDSA
Key Algorithm: [✓ DSA] RSA
Use system generated password: Yes No
Full name: Anthony Wong
Organization unit: Computer science
Organization name: OSU
City/Locality: Stillwater
State/Province: OK
Two-letter country code: US

Figure 4-2. Generate Key Pair dialog box

Delete Key Pair

The **Delete Key Pair** dialog box, as shown in Figure 4-3, is used to remove a key entry, which contains a private key and the associating self-signed certificate, from the user keystore. Once a key entry is deleted, it cannot be recovered. **Key alias** specifies the alias to the key entry. **Key password** specifies the password required to access this key entry.

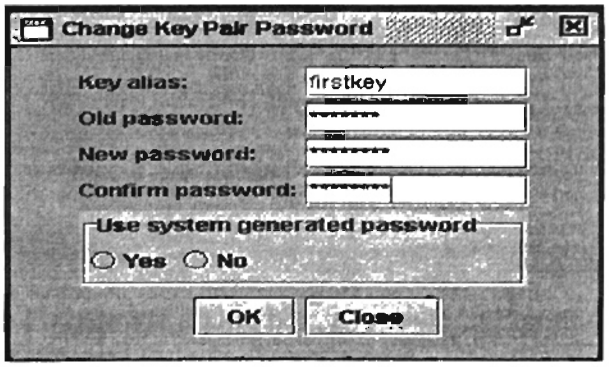


Key alias: firstkey
Key password: ********

Figure 4-3. Delete Key Pair dialog box

Change Key Pair Password

The **Change Key Pair** Password dialog box, as shown in Figure 4-4, is used to change the password associated with a key entry. Key alias specifies the alias to the key entry. **Old password** specifies the password that is currently used to access the key entry. **New password** specifies the new password that will be assigned to the key entry. After the new password is specified, the user needs to reenter the password in **Confirm password**. Use **system generated password** specifies whether or not the user wants to the system to generate a random password for the key entry. Note that this system-generated password uses the random password technique described in chapter 3. The password is eight characters long and consists of ASCII printable characters only (except the space character).



Key alias: firstkey
Old password: ********
New password: *********
Confirm password: *********
Use system generated password
○ Yes ○ No

Figure 4-4. Change Key Pair Password dialog box

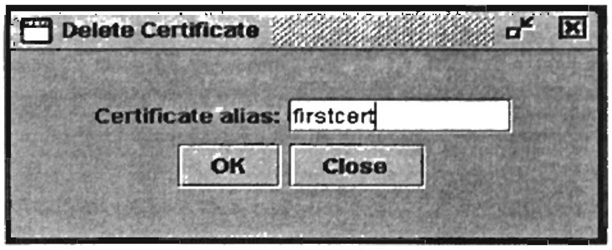
4.3 Certificate

The **Certificate** menus provide services for deleting certificates, importing certificates and certificate chains, exporting certificates, generating CSRs, and printing certificates from certificate files and user keystores. Since the EJST keystore only supports X.509 v1, v2, and v3 certificates, KCPM is limited to operate on these types of certificates. As a consequence, importing, exporting, and printing certificates other than the X.509 standard causes errors.

As mentioned before, a user keystore is capable of storing both key entries and certificate entries. A certificate entry contains a single certificate. It is identified by an alias name just like a key entry does. Note that certificate entries do not have password associating them. This is because certificates are designed to be accessed by the public.

Delete Certificate

The **Delete Certificate** dialog box, as shown is Figure 4-5, is used to remove a certificate entry from a user keystore. **Certificate alias** refers to the alias to the certificate entry.



Certificate alias: firstcert

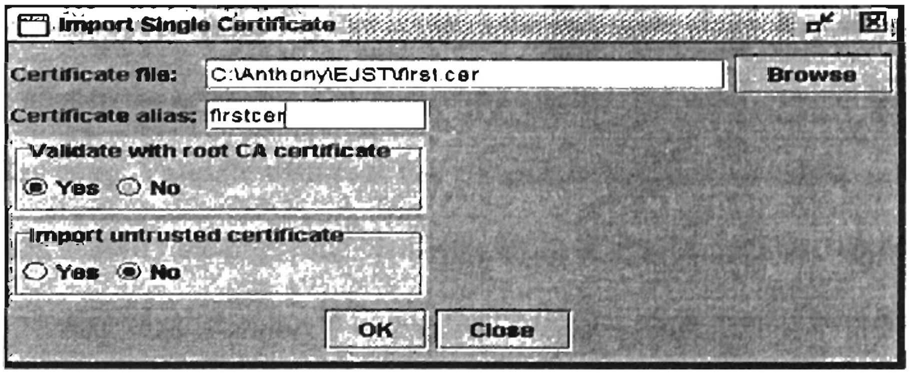
Figure 4-5. Delete Certificate dialog box

Import Single Certificate

The **Import Single Certificate** dialog box, as shown in Figure 4-6, is used to read a certificate from a certificate file and stores it in the user keystore as a certificate entry. If the certificate entry already exists, an error message will be prompted.

KCPM can import both trusted and untrusted certificates to the user keystore. However. importing untrusted certificates is considered dangerous and is not recommended. Adding an untrusted certificate involves no verification. On the other hand. importing a trusted certificate involves series verification operations. KCPM would try to verify the certificate by attempting to construct a chain of trust from that certificate to the certificates from the user keystore. In the chain of trust cannot be established, the certificates from the root CA certificate keystore can be considered. Note that the certificate, which is stored in the certificate file. must be saved in either binary encoding or printable encoding Base 64 format [23, 24].

Certificate file specifies the file that contains the certificate. By convention. certificate files have a "*.cer*" filename extension. **Certificate alias** specifies the alias to the certificate entry. **Validate with root CA certificate** specifies whether or not to use the root CA certificates to verify the importing certificate. **Import untrusted certificate** specifies whether or not to import the certificate even if it is untrusted.


Import Single Certificate

Certificate file: C:\Anthony\EJST\first.cer

Certificate alias: firstcer

Validate with root CA certificate: [✓ Yes] No

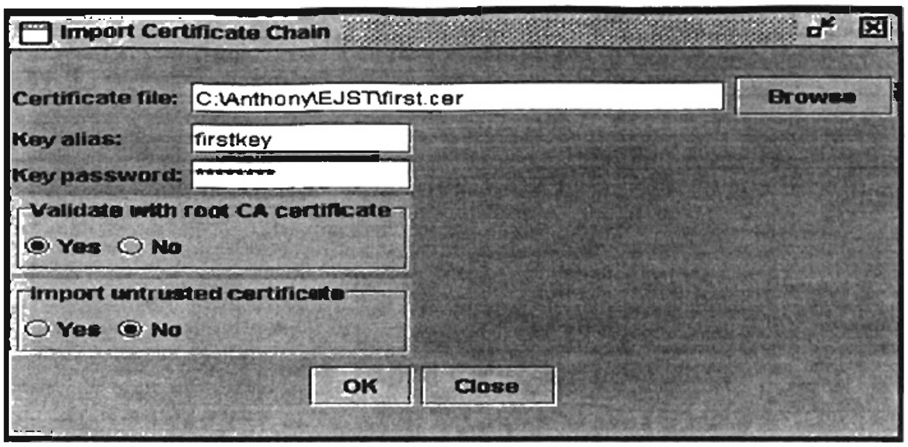
Import untrusted certificate: Yes [✓ No]

Figure 4-6. Import Single Certificate dialog box

Import Certificate Chain

The **Import Certificate Chain** dialog box, as shown in Figure 4-7, is used to read a certificate chain from a certificate file. The certificate chain replaces the certificate chain, which consists only a self-signed certificate and is associated with a private key, in the key entry. The certificate chain from the certificate file should be a PKCS #7 reply, which is generated from a CA in response to a CSR sent by the user. Although KCPM can import X.509 v1, v2, and v3 certificates, the PKCS #7 formatted certificate chain must consist with certificates of the same type [23, 46]. In addition, the certificate chain, which is stored in the certificate file, must be saved in either binary encoding or printable encoding Base 64 format. When importing a certificate reply, the certificate chain can be validated using trusted certificates from the keystore, but this is not required. If the certificate is not validated, the certificate chain from the certificate reply is considered untrusted and it is not recommended. Importing a trusted certificate chain involves series of operations. KCPM tries to verify the certificate by attempting to construct a chain of trust from that certificate to the certificates from the user keystore. If the chain of trust cannot be established, the certificates from the root CA certificate keystore can be considered. Noted that when the certificate chain is imported, it would replace the certificate chain that is associated with the private key in the key entry [23].

Certificate file specifies the file that contains the certificate chain. **Key alias** specifies the alias to the key entry. **Key password** specifies the password required to access this key entry. **Validate with root CA certificate** specifies whether to use the root CA certificates to verify the importing certificate chain or not. **Import untrusted certificate** specifies whether or not to import the certificate chain even if it is untrusted.


Import Certificate Chain

Certificate file: C:\Anthony\EJST\first.cer

Key alias: firstkey

Key password: ********

Validate with root CA certificate: [✓ Yes] No

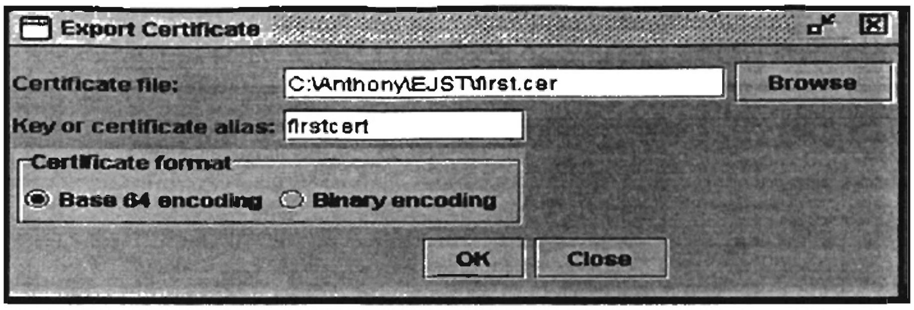
Import untrusted certificate: Yes [✓ No]

Figure 4-7. Import Certificate Chain dialog box

Export Certificate

The **Export Certificate** dialog box, as shown in Figure 4-8, is used to read a certificate from a certificate entry, which is stored in the user keystore, and saved it in a certificate file. By convention, the certificate file has a filename extension "*.cer*". The certificate can be exported either in binary encoding format or printable Base 64 format. If the alias refers to a certificate entry, that certificate is exported. If the alias refers to a key entry, then the first certificate in the associated certificate chain is exported [23, 24].

Certificate file specifies the file that is used to store the certificate. **Key or certificate alias** specifies the alias to a certificate entry or a key entry. **Certificate format** specifies the format that is used to store the certificate. The format can be either binary encoding or printable Base 64 encoding.



Certificate file: C:\Anthony\EJST\first.cer

Key or certificate alias: firstcert

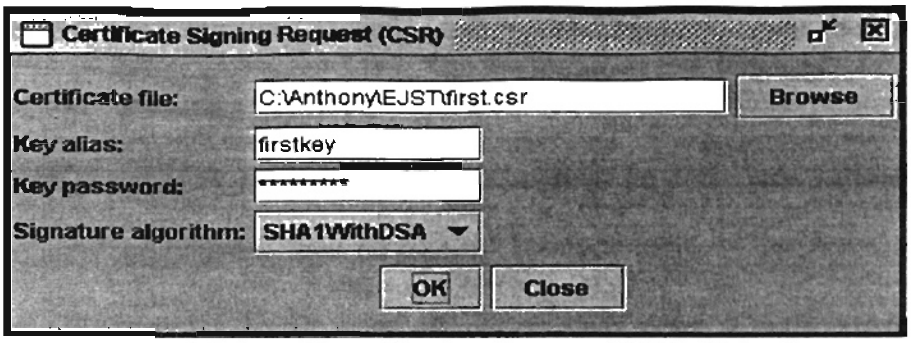
Certificate format: [✓ Base 64 encoding] Binary encoding

Figure 4-8. Export Certificate dialog box

Certificate Signature Request (CSR)

The **Certificate Signing Request** dialog box, as shown in Figure 4-9, is used to generate a CSR using the PKCS #10 format. A CSR is intended to be sent to a CA. The CA will authenticate the certificate requestor and will return a certificate chain in a certificate reply. The certificate requestor will use this certificate chain to replace the existing certificate chain, which is associated with a corresponding private key, in a key entry. The private key and the X.500 distinguished name associated with the key entry are used to create the PKCS #10 CSR [23, 48].

CSR file specifies the file that is used to store the CSR. By convention, a CSR file has an "*.scr*" filename extension. **Key alias** specifies the alias to the key entry. **Key password** specifies the password required to access the key entry. **Signature algorithm** specifies the algorithm used to sign the CSR.



Certificate file: C:\Anthony\EJST\first.csr
Key alias: firstkey
Key password: **********
Signature algorithm: SHA1WithDSA

Figure 4-9. Certificate Signing Request (CSR) dialog box

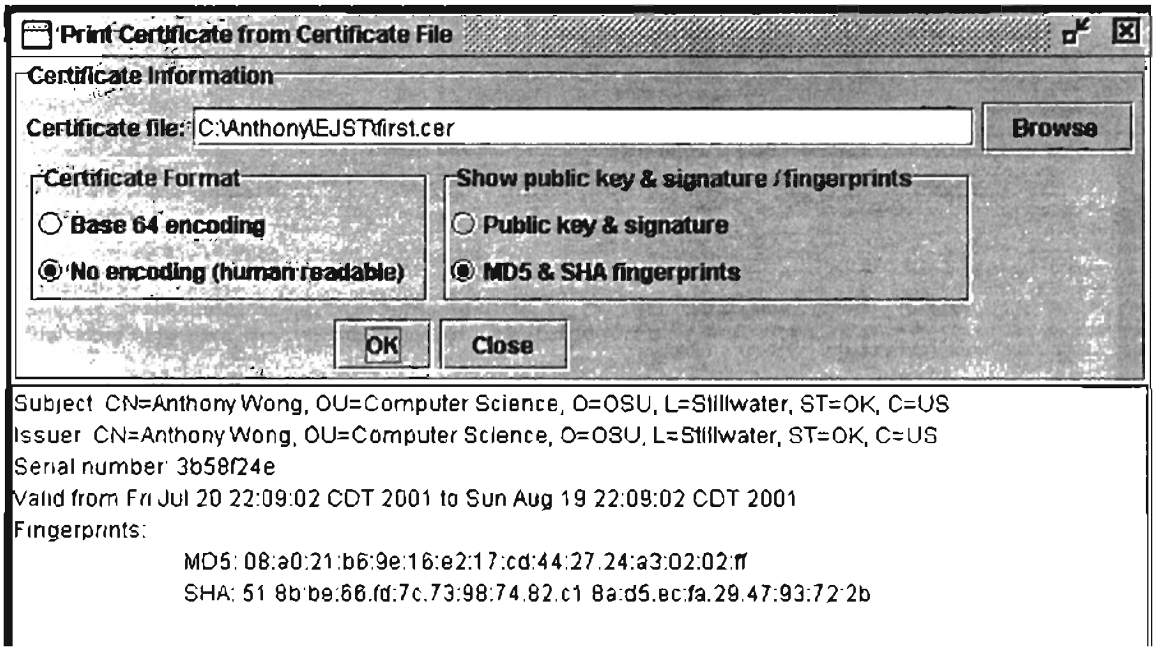
Print Certificate from Certificate File

The **Print Certificate from Certificate File** dialog box, as shown is Figure 4-10, is used to read a certificate from a certificate file and print its content in either printable Base 64 encoding format or using no encoding (human-readable) format. The Printable Base 64 format is used to print the certificate in Base 64 format. The Base 64 format is an Internet standard. Binary data can be encoded in Base 64 by rearranging the bits of the data stream in such a way that only the 6 least significant bits are used in every byte [22, 23].

No encoding with public key and signature format is used to print certificate information that is human readable and understandable. It prints out the alias name of the keystore entry, the certificate creation date and time, the type of the keystore entry, the length of the certificate chain if it is a key entry, the certificate version, the subject in X.500 distinguish name, the name of the signature algorithm used, the name of the public key algorithm, the public key, the time stamp, the issuer in X.500 distinguish name, the serial number, and the value of the signature.

No encoding with fingerprints is used to print certificate information that is human readable and understandable. It prints out the alias name of the keystore entry, the certificate creation date and time, the type of the keystore entry, the length of the certificate chain if it is a key entry, the subject in X.500 distinguish name, the issuer in X.500 distinguish name, the serial number, the time stamp, and the fingerprints of the certificate in MD5 and SHA. Note that the certificate must be stored in either the binary encoding format or the printable Base 64 format.

Certificate file specifies the file where the certificate is stored. **Certificate format** specifies the certificate printout format. If the **no encoding** format is selected, **the Show public key & signature / fingerprints** options will appear. Users can specify their preferred printout option.


Print Certificate from Certificate File

Certificate Information

Certificate file: C:\Anthony\EJST\first.cer

Certificate Format: Base 64 encoding [✓ No encoding (human readable)]

Show public key & signature / fingerprints: Public key & signature [✓ MD5 & SHA fingerprints]

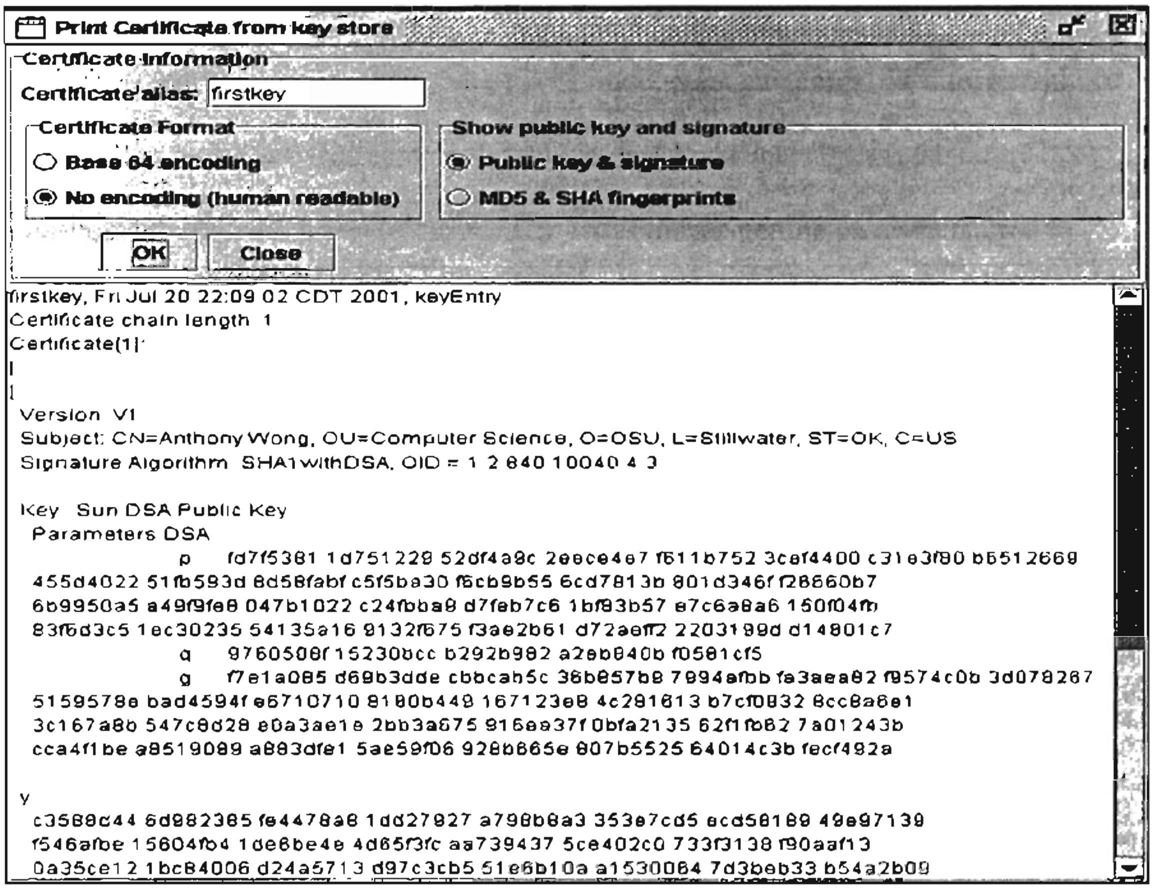
```
Subject  CN=Anthony Wong, OU=Computer Science, O=OSU, L=Stillwater, ST=OK, C=US
Issuer  CN=Anthony Wong, OU=Computer Science, O=OSU, L=Stillwater, ST=OK, C=US
Serial number: 3b58f24e
Valid from Fri Jul 20 22:09:02 CDT 2001 to Sun Aug 19 22:09:02 CDT 2001
Fingerprints:
         MD5: 08:a0:21:b6:9e:16:e2:17:cd:44:27:24:a3:02:02:ff
         SHA: 51:8b:b9:66:fd:7c:73:98:74:82:c1:8a:d5:ec:fa:29:47:93:72:2b
```

Figure 4-10. Print Certificate from Certificate File dialog box

Print Certificate from Keystore

The **Print Certificate from Keystore** dialog box, as shown in Figure 4-11, is used to read a certificate from a certificate entry or key entry and prints out its content in either printable Base 64 encoding format or using no encoding (human-readable) format. The no encoding format further specifies the details on the printout: public key and signature, or MD5 and SHA fingerprints. Note that the certificate must be stored in either the binary encoding format or the printable Base 64 format.

Certificate file specifies the file, which the certificate is stored. **Certificate format** specifies the certificate printout format. If the **no encoding** format is selected, the **Show public key & signature / fingerprints** options will appear. Users can specify their prefer printout option.


Print Certificate from key store
Certificate Information
Certificate alias: firstkey
Certificate Format
Base 64 encoding
No encoding (human readable)
Show public key and signature
Public key & signature
MD5 & SHA fingerprints
OK
Close

Certificate Format: Base 64 encoding [✓ No encoding (human readable)]

Show public key and signature: [✓ Public key & signature] MD5 & SHA fingerprints

```
firstkey, Fri Jul 20 22:09 02 CDT 2001, keyEntry
Certificate chain length 1
Certificate[1]:
[
[
 Version V1
 Subject: CN=Anthony Wong, OU=Computer Science, O=OSU, L=Stillwater, ST=OK, C=US
 Signature Algorithm SHA1withDSA, OID = 1 2 840 10040 4 3

 Key  Sun DSA Public Key
  Parameters DSA
                 p     fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
  455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
  6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
  83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
                 q     9760508f 15230bcc b292b982 a2eb840b f0581cf5
                 g     f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
  5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
  3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
  cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

y
  c3588c44 6d982385 fe4478a8 1dd27827 a798b8a3 353e7cd5 ecd58189 49e97139
  f546afbe 15604fb4 1de6be4e 4d65f3fc aa739437 5ce402c0 733f3138 f90aaf13
  0a35ce12 1bc84006 d24a5713 d97c3cb5 51e6b10a a1530084 7d3beb33 b54a2b09
```

Figure 4-11. Print Certificate from Keystore dialog box

4.4 Sign / Verify JAR

KPCM provides services for signing JAR files and verifying signatures and the integrity of signed JAR files. In addition, it also provides services for signing regular files (non JAR files.)

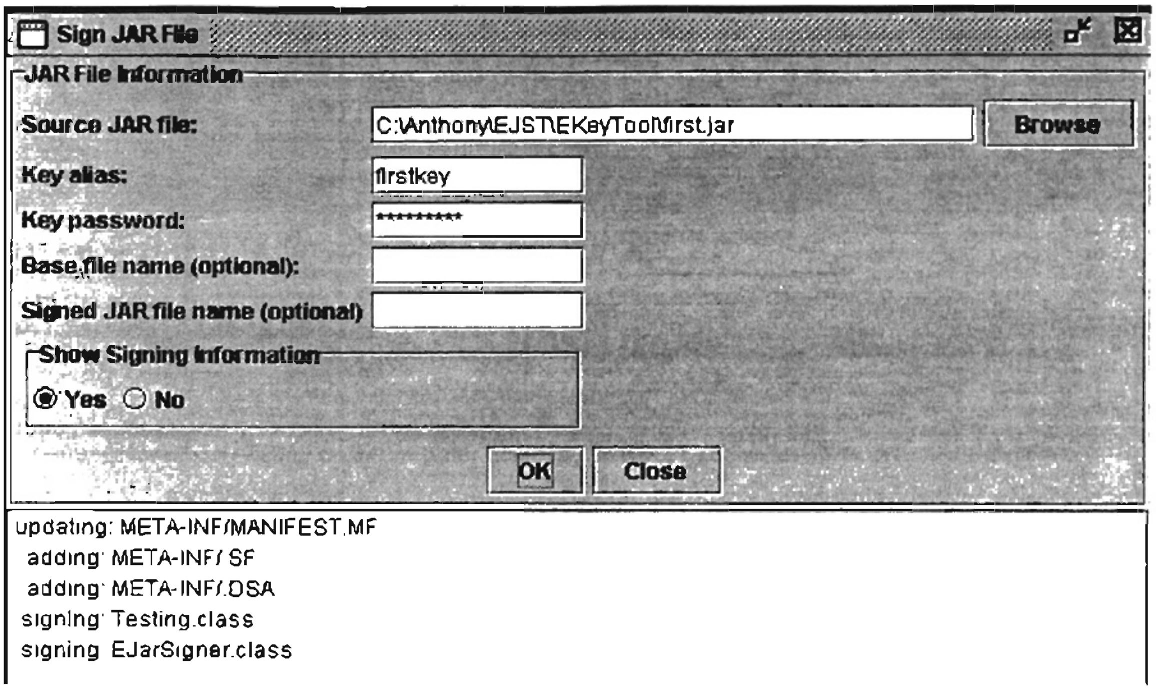
A signed JAR file contains a signature file, with a *SF* extension, and a signature block file, with a *DSA* extension. Each *SF* file contains three lines of text: the signature version, the name of the digest algorithm for the manifest file and the digest value for the manifest file, and the name of the company that provides the digest algorithm. In addition, the *SF* file contains two lines of text for each source file archived in the JAR file. The first line specifies the name of the source file, and the second line specifies the name of the digest algorithm and the digest value. The *SF* file is then signed. Each *DSA* signature block file, by default, contains the same three line of text found in the associated *SF* file. In addition, the *DSA* file contains the signature of the *SF* file. It also contains the encoded certificate or certificate chain from a user keystore. The certificate or certificate chain is used to authenticate the public key corresponding to the private key used for signing [23, 24].

Sign JAR File

The **Sign JAR File** dialog box, as shown in Figure 4-12, is used to sign a JAR file. The signing process produces a signature file with a *SF* extension and a signature block file with a *DSA* extension. KCPM uses the private key from a key entry in a user keystore to sign the *SF* file and the associating certificate chain to provide authenticity.

Source JAR file specifies the JAR file to be signed. **Key alias** specifies the alias to the key entry. **Key password** specifies the password required to access this key entry. **Base file name** specifies the base file name for the *SF* and *DSA* file. Note that this is optional. **Signed JAR file** specifies the name of the signed JAR file. This is optional too. If one does not specify a file name, the source JAR file is used as the signed JAR file; thus the content of the source JAR file will be overwritten. **Show signing information**

specifies whether a detail description of the signing process should be printed after the file is signed.



Source JAR file: C:\Anthony\EJST\EKeyTool\first.jar

Key alias: firstkey

Key password: *********

Base file name (optional):

Signed JAR file name (optional)

(● Yes ○ No)

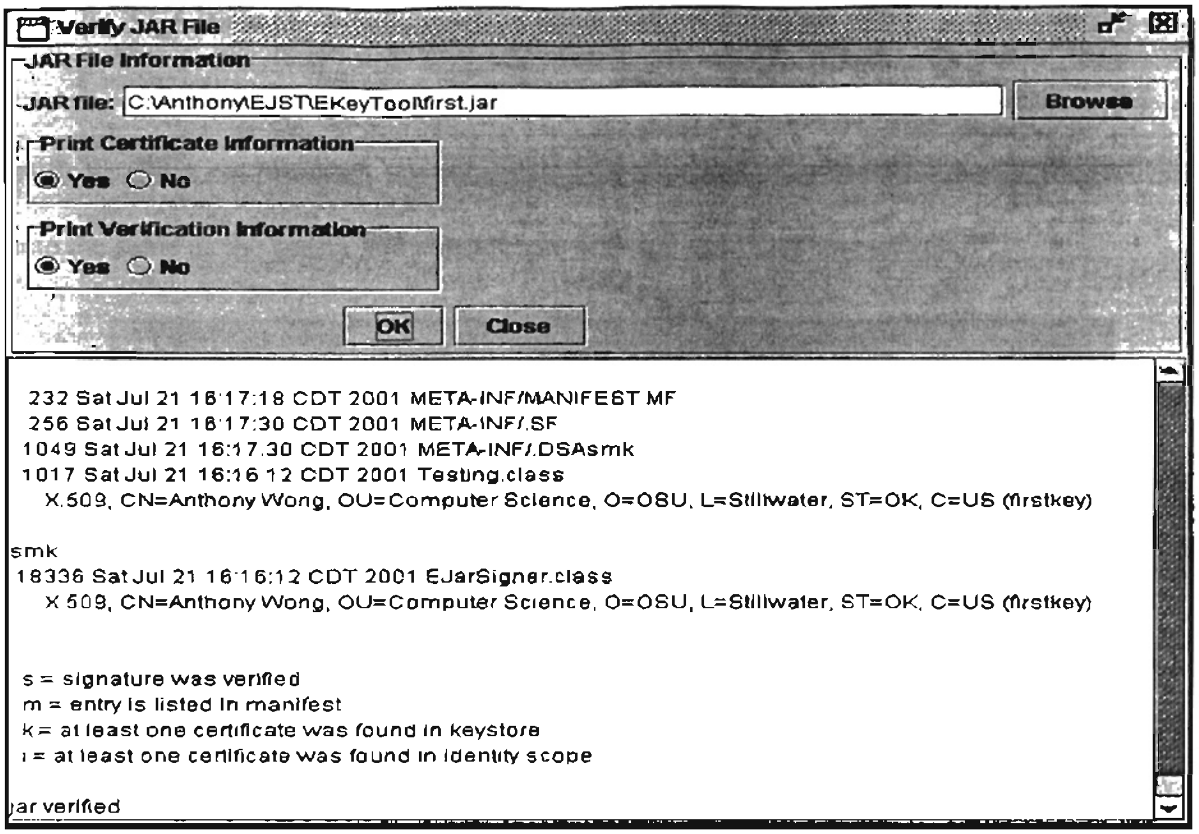
```
updating: META-INF/MANIFEST.MF
 adding: META-INF/.SF
 adding: META-INF/.DSA
signing: Testing.class
signing: EJarSigner.class
```

Figure 4-12. Sign JAR File dialog box

Verify JAR File

The **Verify JAR File** dialog box, as shown in Figure 4-13, is used to verify signatures and the integrity of signed JAR files. The integrity check will print an error message if any of the files in the JAR file were modified. If the JAR file is not signed, an error message will also be printed out.

JAR file specifies the signed source JAR file. **Print certificate information** specifies whether or not the certificate stored in the *DSA* file be printed out after the verification process. **Print verification information** specifies whether or not a detailed message is printed after the verification process.



JAR file: C:\Anthony\EJST\EKeyTool\first.jar

Print Certificate Information: [✓ Yes] No

Print Verification Information: [✓ Yes] No

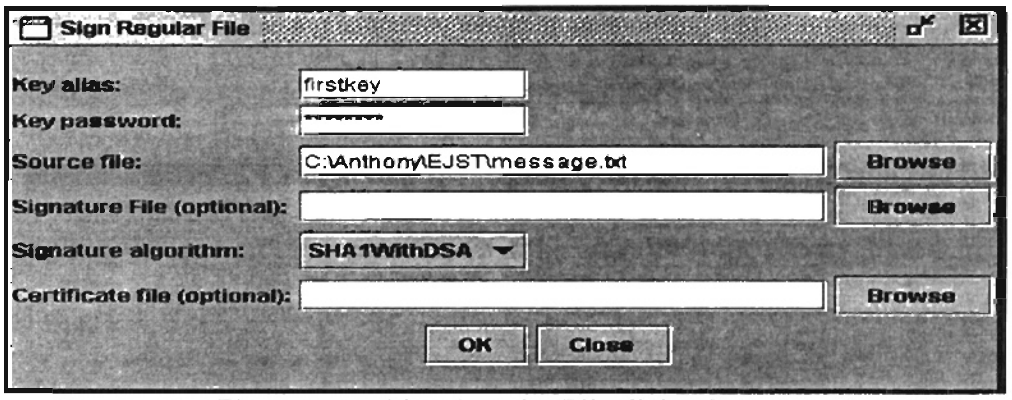
```
 232 Sat Jul 21 16:17:18 CDT 2001 META-INF/MANIFEST.MF
 256 Sat Jul 21 16:17:30 CDT 2001 META-INF/.SF
1049 Sat Jul 21 16:17:30 CDT 2001 META-INF/.DSAsmk
1017 Sat Jul 21 16:16:12 CDT 2001 Testing.class
   X.509, CN=Anthony Wong, OU=Computer Science, O=OSU, L=Stillwater, ST=OK, C=US (firstkey)

smk
18336 Sat Jul 21 16:16:12 CDT 2001 EJarSigner.class
   X.509, CN=Anthony Wong, OU=Computer Science, O=OSU, L=Stillwater, ST=OK, C=US (firstkey)


 s = signature was verified
 m = entry is listed in manifest
 k = at least one certificate was found in keystore
 i = at least one certificate was found in identity scope

jar verified
```

Figure 4-13. Verify JAR File dialog box

Sign Regular File

The **Sign Regular File** dialog box, as show in Figure 4-14, is used to sign all kinds of files except the class and JAR files. The signing process uses the private key in a key entry to sign the source file. After the source file is signed, a signature file and a certificate file are produced. The signature file and certificate file are used to protect the data integrity of the signed file.

Key alias specifies the alias to the key entry. **Key password** specifies the password required to access the key entry. **Source file** specifies the source message file to be signed. **Signature file** specifies the signature file that is use to store the digital signature of the source file. If the signature file is not provided, the signature will be stored in a file with the base file name the same as the source file concatenate with a "*.sig*" filename extension. **Signature algorithm** specifies the signature algorithm that is used to sign the message. **Certificate file** specifies the file that is used to store the digital certificate. If the certificate file is not provided, the certificate will be stored in a file with the base file name the same as the source file concatenate with a "*.cer*" filename extension.



Sign Regular File

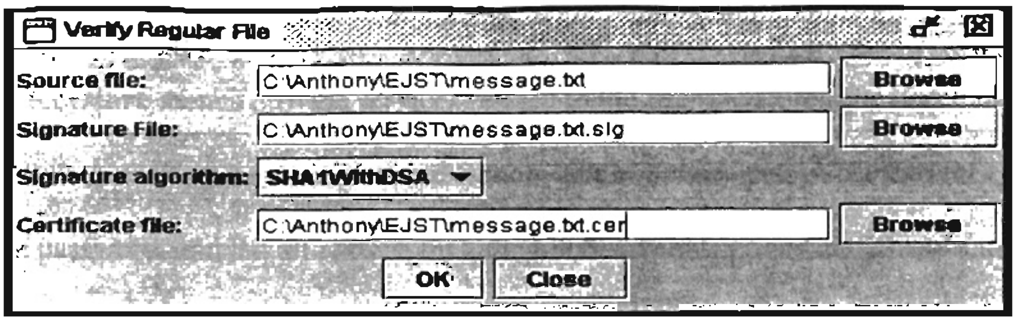
Key alias:	firstkey	
Key password:	********	
Source file:	C:\Anthony\EJST\message.txt	Browse
Signature File (optional):		Browse
Signature algorithm:	SHA1WithDSA	
Certificate file (optional):		Browse

Figure 4-14. Sign Regular File dialog box

Verify Regular File

The **Verify Regular** File dialog box, as shown in Figure 4-15, is used to verify signatures and the integrity of signed files. Unlike signing regular files, users need to specify the original source file (signed file), the signature file, and the certificate file.

Source file specifies the original (signed) file. **Signature file** specifies the file that is used to store the signature associated with the source file. **Signature algorithm** specifies the signature algorithm used in signing the source file. **Certificate file** specifies the file that is used to store the certificate associated with the source file.



C:\Anthony\EJST\message.txt

C:\Anthony\EJST\message.txt.sig

SHA1WithDSA

C:\Anthony\EJST\message.txt.cer

Figure 4-15. Verify Regular File dialog box

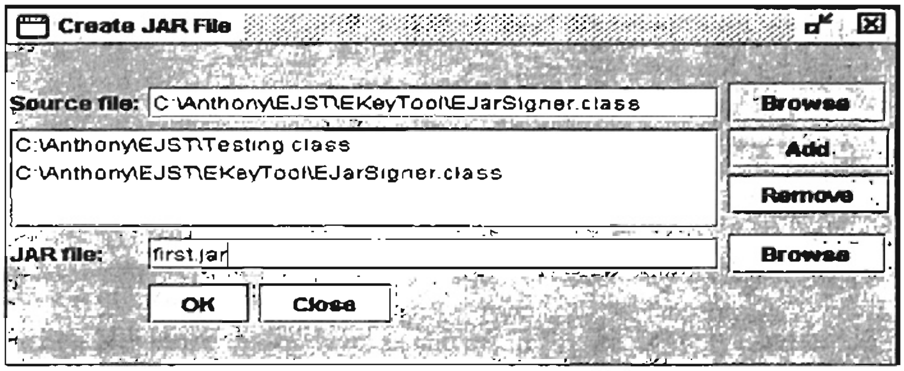
4.5 JAR

KPCM provides services on archiving files through the JAR menus. A *manifest* file is created during the JAR file creation process. This file contains information about each of the archived files [23, 24].

Create JAR File

The **Create JAR File** dialog box, as shown in Figure 4-16, is used to create JAR files. First, the source files to be archived are selected. Next the files are added to the archive list. After specifying the name of the destination JAR file, the JAR file is created.

Source file specifies a file to be archived. The **Add** button adds the source files to the **archive file list**. The **remove** button removes the selected files from the **archive file list. JAR file** specifies the file that holds the archived source files and the *manifest* file.



C:\Anthony\EJST\EKeyTool\EJarSigner.class

C:\Anthony\EJST\Testing class
C:\Anthony\EJST\EKeyTool\EJarSigner.class

first.jar

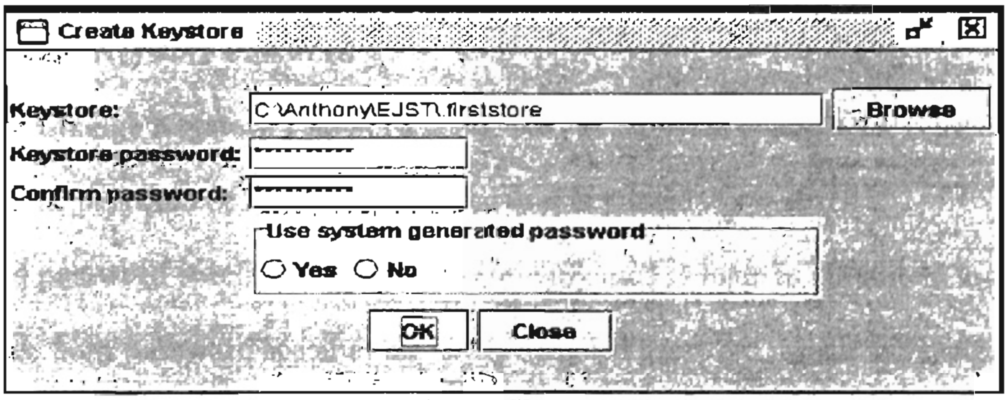
Figure 4-16. Create JAR File dialog box

4.6 Keystore

The **Keystore** menus provide services related to the user keystore. The user can create a new user keystore, change to a different user keystore, change the current keystore password, and list all entries, including both key entries and certificate entries, from the current user keystore.

Create Keystore

The **Create Keystore** dialog box, as shown in Figure 4-17, is used to create a new user keystore. The newly created keystore does not contain any entry. **Keystore** specifies the file used to store the user keystore. **Keystore password** specifies the password required to access the user keystore. The user needs to re-enter the password to for confirmation in **Confirm password**. **Use system generated password** option specifies whether or not the user wants KCPM to generate a random password for the keystore password.



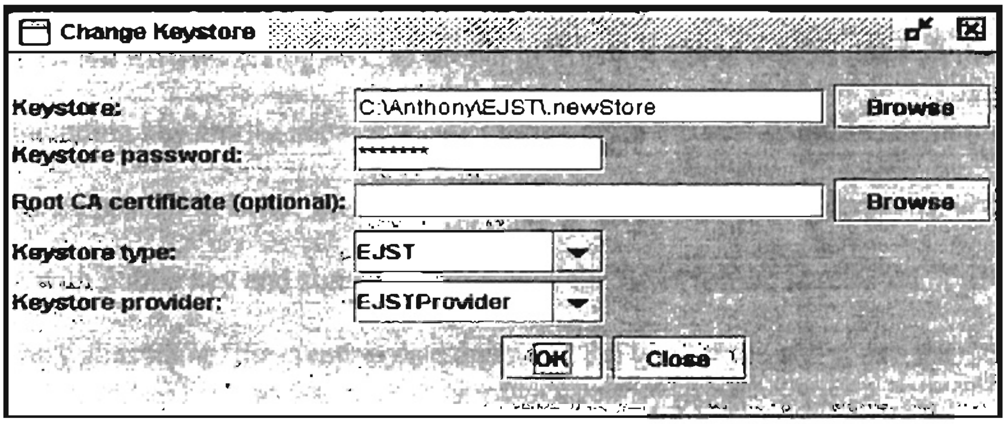
Keystore: C:\Anthony\EJST\.firststore
Keystore password: ***********
Confirm password: ***********
Use system generated password: ○ Yes ○ No

Figure 4-17. Create Keystore dialog box

Change keystore

The **Change Keystore** dialog box, as shown in Figure 4-18, is used to load a different user keystore to the system. Users also can load a different root CA certificate keystore using this dialog box. After the new user keystore is loaded, all the keystore management and signature signing and verification operations will refer to the newly loaded user keystore.

Keystore specifies the keystore file. **Keystore password** specifies the password required to access the user keystore. Note that if the user does not specify the root CA certificate keystore, the previous defined root CA certificate keystore is used. **Keystore type** specifies the type of the keystore implementation. **Keystore provider** specifies the keystore implementation service provider (creator).



Keystore: C:\Anthony\EJST\.newStore
Keystore password: *******
Root CA certificate (optional):
Keystore type: EJST
Keystore provider: EJSTProvider

Figure 4-18. Change Keystore dialog

Change Keystore Password

The **Change Keystore Password** dialog box, as shown in Figure 4-19, is used to change the password of the currently loaded user keystore.

Old password specifies the password currently used to login to the current user keystore. **New password** specifies the newly assigned password, which will replace the old password. The user is required to re-enter the new password in **Confirm password** to confirm the password that has been entered. **Use system generated password** specifies whether or not the user wants KCPM to generate a random password for the keystore.

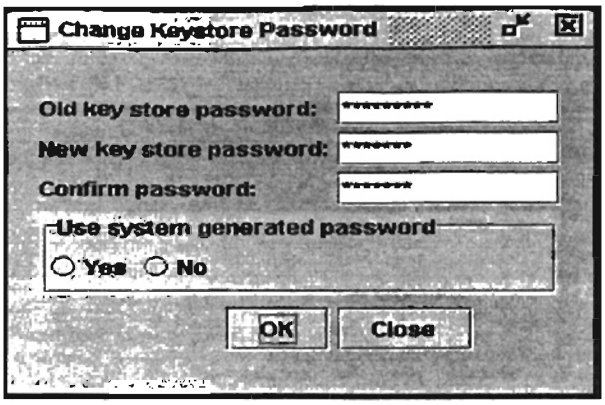
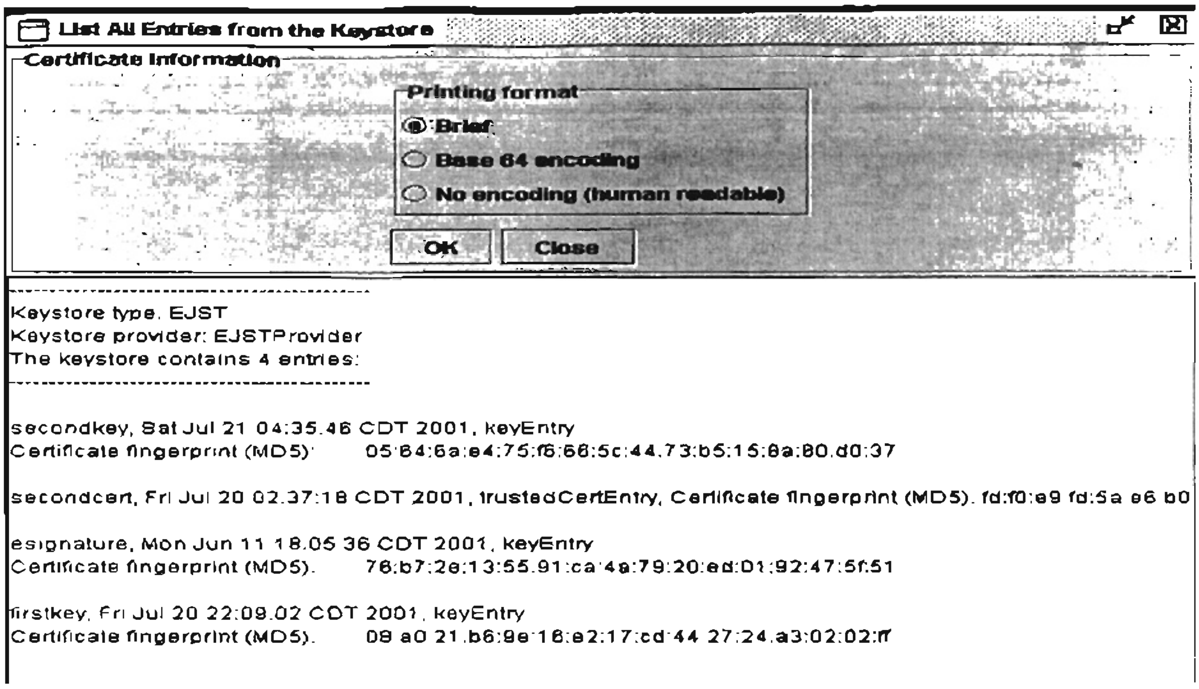


Figure 4-19. Change Keystore Password dialog box

List All Entries

The **List All Entries** dialog box, as shown in Figure 4-20, is used to print all the entries, including key entries and certificate entries, from the user keystore. Keystore entries can be printed in Brief, Base 64 encoding, no encoding with fingerprints, and no encoding with public key and signature formats. If the Brief format is selected, it prints out the entry alias name, the creation date and time, the type of the entry, and the certificate fingerprint in MD5. If the entry is a key entry, the fingerprint of the first certificate in the certificate chain is printed. Base 64 encoding, no encoding with public key and signature, and no encoding with fingerprints all print out entry alias name, the creation date and time, the type of the entry, and the length of the certificate chain, if the entry is a key entry. In addition, information about the certificate or certificate chain will also be printed as discussed early in this menu **Printing format** specifies the keystore entry printout format as described above.



Printing format: [✓ Brief] Base 64 encoding No encoding (human readable)

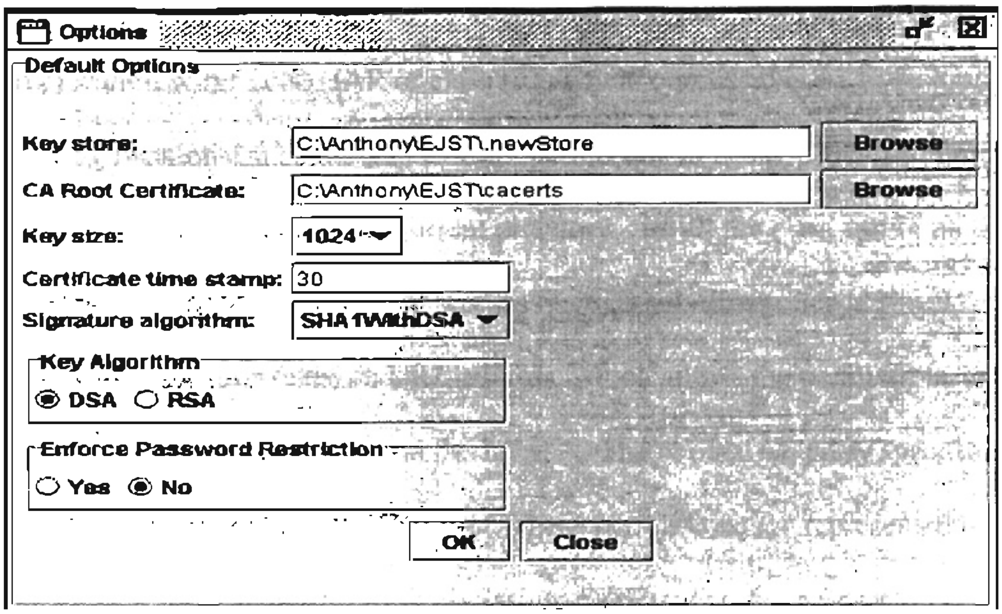
```
------------------------------------------
Keystore type. EJST
Keystore provider: EJSTProvider
The keystore contains 4 entries:
------------------------------------------

secondkey, Sat Jul 21 04:35.46 CDT 2001, keyEntry
Certificate fingerprint (MD5)'      05:64:6a:e4:75:f6:66:5c:44.73:b5:15:6a:80.d0:37

secondcert, Fri Jul 20 02.37:18 CDT 2001, trustedCertEntry, Certificate fingerprint (MD5). fd:f0:e9 fd:5a e6 b0

esignature, Mon Jun 11 18.05 36 CDT 2001, keyEntry
Certificate fingerprint (MD5).      76:b7:2e:13:55.91:ca'4a:79:20:ed:01:92:47:5f:51

firstkey, Fri Jul 20 22:09.02 CDT 2001, keyEntry
Certificate fingerprint (MD5).      08 a0 21.b6:9e'16:e2:17:cd'44 27:24.a3:02:02:ff
```

Figure 4-20. List All Entries dialog box

4.7 Options

The **Options** dialog box, as shown in Figure 4-21, is used to provide default configuration options for KCPM. These configurations are used throughout the application, and they are stored in a file named *option.txt*. When KCPM is started, it reads the *option.txt* file to initialize the configurations.

Keystore specifies the default file path of the user keystore. **Root CA certificate** specifies the default file path of the root CA certificate keystore. **Key size** specifies the default value of the size of the keys used in the public-key cipher. **Certificate time stamp** specifies the default of the time stamp, which is the validity of a certificate in days. **Signature algorithm** specifies the default signature algorithm. **Enforce password restriction** specifies whether or not password restrictions be applied. These restrictions are applied to both user keystore passwords and key entry passwords. If enforced, passwords provided by the users must be at least six letters long, and they must consist of one or more punctuation or numeric characters.


Options

Default Options

Key store: C:\Anthony\EJST\.newStore

CA Root Certificate: C:\Anthony\EJST\cacerts

Key size: 1024

Certificate time stamp: 30

Signature algorithm: SHA1WithDSA

Key Algorithm: [✓ DSA] RSA

Enforce Password Restriction: Yes [✓ No]

Figure 4-21. Options dialog box

4.8 Security Policy

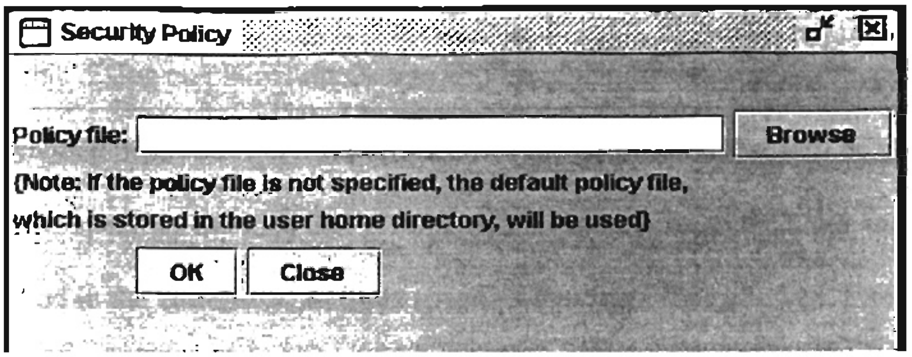
Java's security policies are used to grant permissions to various Java code (class and JAR files) depending upon the code base and/or digital signature applied to the code. These security policies are stored in a security policy file. Java 2 comes with a policy file named *java.policy*, and it is stored in the *java.home\lib\security* directory [23].

A policy file contains a list of entries. There may be a *keystore* entry and zero or more *grant* entries. There can be only one keystore entry on a policy file. This entry is necessary only if a signer is specified in any of the *grant* entries so that the signer can be referred to the specific keystore [23].

Grant entries are used to grant permissions to codes from various sources and/or signed by various entities. Each *grant* entry contains zero or more *permission* entries, and can include a *signedBy* name-value pair entry and a *codeBase* name-value pair entry. The *signedBy* entry specifies that the permissions granted are for the code that has been signed by the private key from the corresponding key entry. When multiple signers are specified, the code must be signed by all of them. The *codeBase* entry specifies the originating location of the code (JAR or class file) where permissions are to be granted. The originating location can be defined to grant permission to a single JAR or class file, all class and/or all JAR files from the current directory, and all the class and/or all the JAR files from the currently directory as well as from its sub-directories [23].

The originating location takes an URL address, which allows the permissions to be granted to different networks that are on the Internet. The *permission* entry specifies permissions that are granted to specific target. These permissions are represented by permission classes. They include: *AWTPermission*, *FilePermission*, *NetPermission*, *PropertyPermission*, *ReflectPermission*, *RuntimePermission*, *SecurityPermission*, *SerializablePermission*, and *SocketPermission*. The *target* is the files that give particular permissions to the class and/or JAR file specified in the *codeBase*. Creating policies on policy files is outside the scope of this paper. For more information about the security policy, please refer to [22], [23], and [24].

The **Security Policy** dialog box, as shown in Figure 4-22, is used to connect to the ***policytool*** provided by Java 2. **Policy files** specifies the security policy file that will be accessed by the ***policytool***. If the policy file is not specified, ***policytool*** will try to open the policy file in the *user.home* directory. If the policy file does not exist, the user will be presented with an error message.



(Note: If the policy file is not specified, the default policy file, which is stored in the user home directory, will be used)

Figure 4-22. Security Policy dialog box

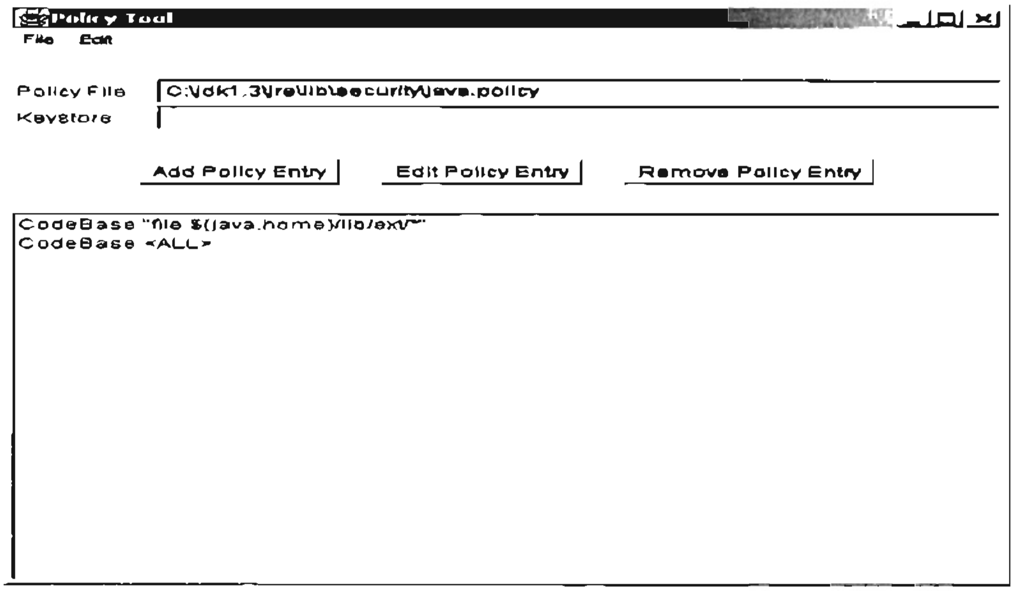
The **Policy Tool** dialog box, as shown in Figure 4-21, is designed to configure security policies on Java applications and applets in a user computer. It provides three functions: create new policies, modify existing policies, and view warning logs.

To create a new policy, first, make sure that the policy file is not under construction. If the ***policytool*** is working on an existing policy file, select **new** from the **File** menu. This will open a new policy. Then select **save** from the **File** menu to save the policy to a policy file. The file should be saved with a *policy* extension.

The ***policytool*** maintains a warning log, which stored all the warning messages that have been displayed during a policy configuration session. To access this warning log, **select view warning log** from the **File** menu.

The ***policytool*** allows adding new policy entries, modifying existing policy entries, removing existing policy entries, and changing to another user keystore.

To add a new policy entry, click on the **Add Policy Entry** button on the **Policy Tool** dialog box, as shown in Figure 4-23. This brings up the **Policy Entry** dialog box, as shown in Figure 4-24. **codeBase** specifies the originate location of the code, which permissions are to be granted. **signedBy** specifies the alias name to the key entry. To add permission to this policy entry, click on the **Add Permission** button. This will bring up the **Permissions** dialog box, as shown in Figure 4-25. The drop-down lists (combo boxes) on this dialog box allow the user to choose among the various options that are already provided in Java 2. **Permission** specifies the permission that will be granted. **Target name** specifies the files that give particular permission to the class and/or JAR files specified in the **codeBase**. **Actions** specifies the operations allowed. FilePermission, for example, can have read, write, delete, and execute operations. To edit permissions, click on the **Edit Permission** button. To remove permissions, click on the **Remove Permission** button [23].



Policy File C:\jdk1.3\jre\lib\security\java.policy
Keystore

```
CodeBase "file ${java.home}/lib/ext/*"
CodeBase <ALL>
```

Figure 4-23. Policy Tool dialog box

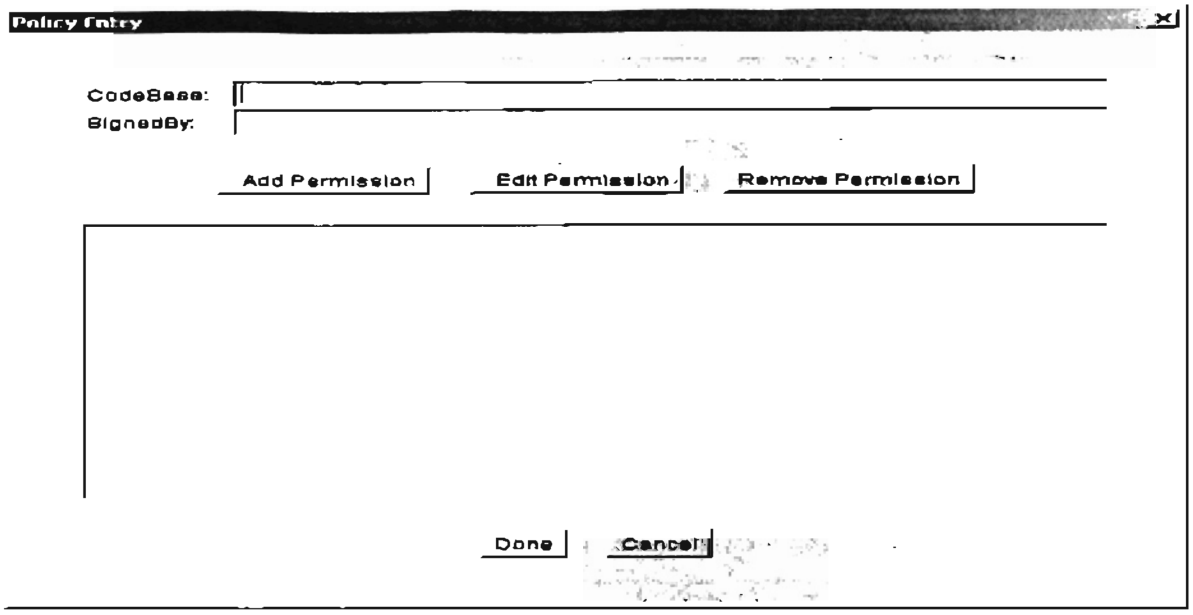
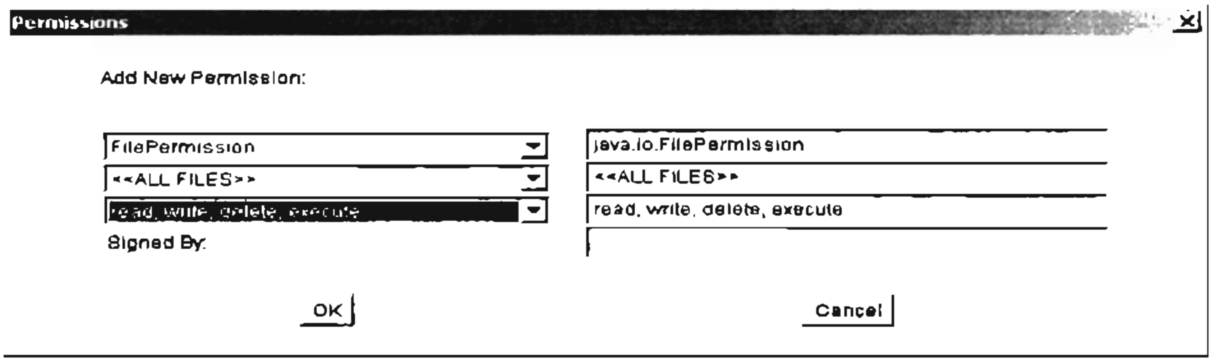


Figure 4-24. Policy Entry dialog box



java.io.FilePermission
<<ALL FILES>>
read, write, delete, execute

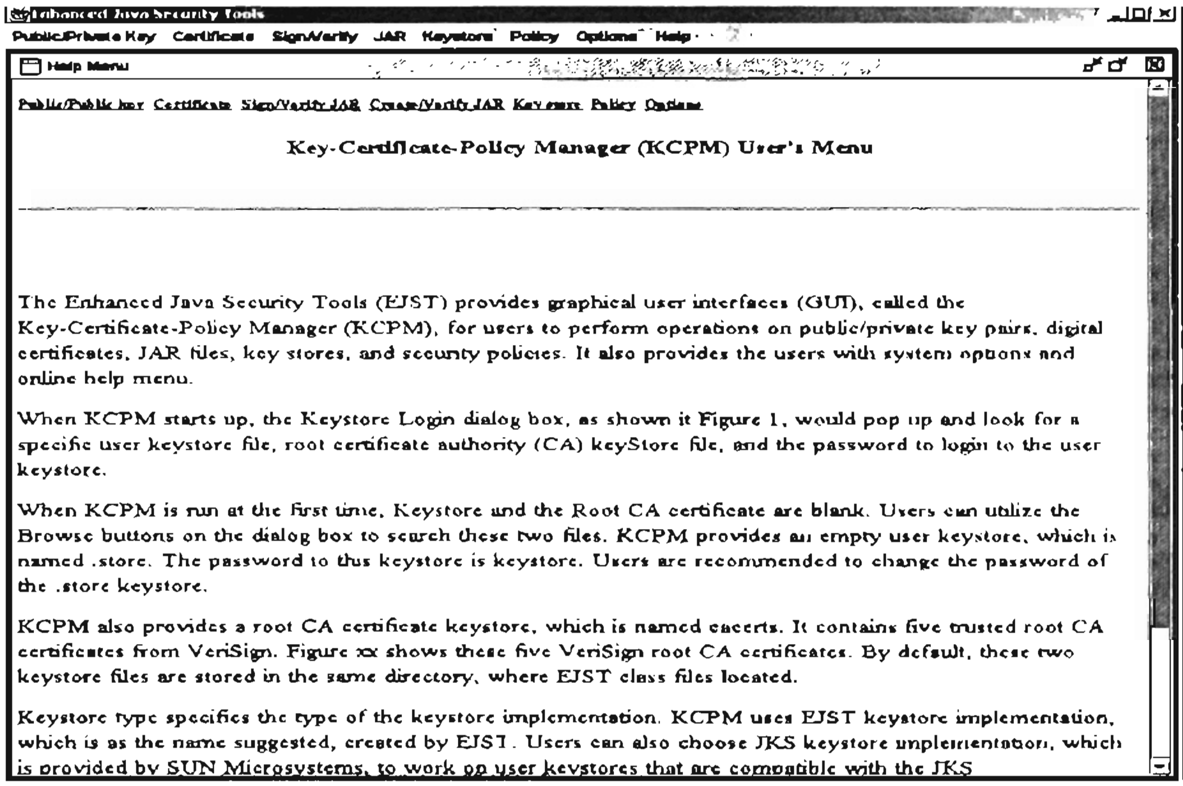
Figure 4-25. Permissions dialog box

To edit an existing policy entry, click on the **Edit Policy Entry** button on the **Policy Tool** dialog box. This brings up the **Policy Entry** dialog box. Modification of an existing policy entry can be done in the same way as adding new policy entries that is described above.

To change the user keystore so that the policy configuration should apply, select **Change keystore** from the **Edit** menu. The **Keystore** dialog box will pop up. **New keystore** URL specifies the location of the user keystore file. **New keystore type** specifies the type of the implementation used in the user keystore. The keystore supplied by Java 2 is implemented by Sun Microsystems. This keystore has a type called *JKS*. KCPM has its own keystore implementation, and it is of the type *EJST*.

4.9 Help Menu

The **Help Menu** dialog box, as shown in Figure 4-26, is used to display KCPM's help menu. This help menu provides detailed information about every service available on KCPM. The **Help Menu** dialog is designed like a web browser. Users can browse the help menu through links.


Enhanced Java Security Tools
Public/Private Key Certificate Sign/Verify JAR Keystore Policy Options Help
Help Menu
Public/Public key Certificate Sign/Verify JAR Create/Verify JAR Keystore Policy Options

Key-Certificate-Policy Manager (KCPM) User's Menu

The Enhanced Java Security Tools (EJST) provides graphical user interfaces (GUI), called the Key-Certificate-Policy Manager (KCPM), for users to perform operations on public/private key pairs, digital certificates, JAR files, key stores, and security policies. It also provides the users with system options and online help menu.

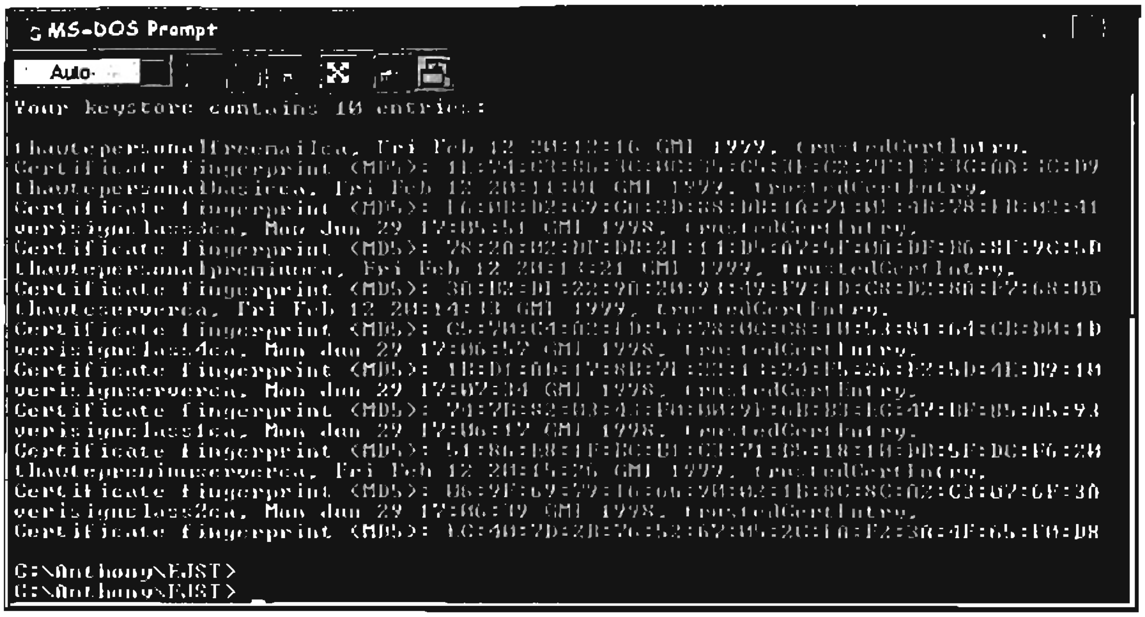
When KCPM starts up, the Keystore Login dialog box, as shown it Figure 1, would pop up and look for a specific user keystore file, root certificate authority (CA) keyStore file, and the password to login to the user keystore.

When KCPM is run at the first time, Keystore and the Root CA certificate are blank. Users can utilize the Browse buttons on the dialog box to search these two files. KCPM provides an empty user keystore, which is named .store. The password to this keystore is keystore. Users are recommended to change the password of the .store keystore.

KCPM also provides a root CA certificate keystore, which is named cacerts. It contains five trusted root CA certificates from VeriSign. Figure xx shows these five VeriSign root CA certificates. By default, these two keystore files are stored in the same directory, where EJST class files located.

Keystore type specifies the type of the keystore implementation. KCPM uses EJST keystore implementation, which is as the name suggested, created by EJST. Users can also choose JKS keystore implementation, which is provided by SUN Microsystems, to work on user keystores that are compatible with the JKS

Figure 4-26. Help Menu dialog box



```
Your keystore contains 10 entries:

thawtepersonalfreemailca, Fri Feb 12 20:12:16 GMT 1999, trustedCertEntry,
Certificate fingerprint (MD5): 1E:74:C3:86:3C:0C:35:C5:3E:C2:7F:EF:3C:AA:3C:D9
thawtepersonalbasicca, Fri Feb 12 20:11:01 GMT 1999, trustedCertEntry,
Certificate fingerprint (MD5): E6:0B:D2:C9:CA:2D:88:DB:1A:71:0E:4B:78:EB:02:41
verisignclass3ca, Mon Jun 29 17:05:51 GMT 1998, trustedCertEntry,
Certificate fingerprint (MD5): 78:2A:02:DF:DB:2E:14:D5:A7:5F:0A:DF:B6:8E:9C:5D
thawtepersonalpremiumca, Fri Feb 12 20:13:21 GMT 1999, trustedCertEntry,
Certificate fingerprint (MD5): 3A:B2:DE:22:9A:20:93:49:F9:ED:C8:D2:8A:E7:68:0D
thawteserverca, Fri Feb 12 20:14:33 GMT 1999, trustedCertEntry,
Certificate fingerprint (MD5): C5:70:C4:A2:ED:53:78:0C:C8:10:53:81:64:CB:D0:1D
verisignclass4ca, Mon Jun 29 17:06:57 GMT 1998, trustedCertEntry,
Certificate fingerprint (MD5): 1B:D1:AD:17:8B:7F:22:13:24:F5:26:E2:5D:4E:B9:10
verisignserverca, Mon Jun 29 17:07:34 GMT 1998, trustedCertEntry,
Certificate fingerprint (MD5): 74:7B:82:03:43:F0:00:9E:6B:B3:EC:47:BF:85:A5:93
verisignclass1ca, Mon Jun 29 17:06:17 GMT 1998, trustedCertEntry,
Certificate fingerprint (MD5): 51:86:E8:1F:BC:B1:C3:71:B5:18:10:DB:5F:DC:F6:20
thawtepremiumserverca, Fri Feb 12 20:15:26 GMT 1999, trustedCertEntry,
Certificate fingerprint (MD5): 06:9F:69:79:16:66:90:02:1B:8C:8C:A2:C3:07:6F:3A
verisignclass2ca, Mon Jun 29 17:06:39 GMT 1998, trustedCertEntry,
Certificate fingerprint (MD5): EC:40:7D:2B:76:52:67:05:2C:EA:F2:3A:4F:65:F0:D8

C:\Anthony\EJST>
C:\Anthony\EJST>
```

Figure 4-27. VeriSign root CA certificates

5 CONCLUSIONS

Java's security model provides security APIs and security tools for key management, digital signatures, digital certificates, access control, and more. Although the security APIs and the security tools have been improved, security flaws, weaknesses, and limitations still remain. These problems involve keystore key management, pseudo-random number generation, and the security tools.

The Enhanced Java Security Tools (EJST) provides solutions to these problems. To seal the loopholes, EJST implemented the Truly Random Seeders (TRS) and the Secure Keystore. TRS provides truly random seeds that can be used on any pseudo-random number generator to reduce or eliminate patterns. The Secure Keystore utilizes the PBEWithMD5AndDES password-based encryption algorithm to provide a more secure key storage scheme.

EJST not only fixes the security loopholes, but also provides enhancements on the Java security tools. The Key-Certificate-Policy Manager (KCPM) is implemented to replace the command line user interfaces for the Java security tools. It provides GUIs that are user-friendly. It contains an online help menu to aid the users. It embeds the password restriction and random password techniques to provide stronger authentication for the underlying keystore. Furthermore, it extends data integrity checking to all file types.

EJST is a multi-platform application. It can be installed to any computer that supports Java 2. With a few configurations, EJST can be up and running.

6 FUTURE WORKS

Although the Enhanced Java Security Tools (EJST) has made many enhancements on the Java security tools, there is still room for improvement.

First of all, keystores should not be limited for local access. They should be designed for remote access as well. An organization may want to install a keystore on their network, so that members of the network can enjoy the services provided by the distributed keystore. A distributed keystore can be placed inside a remote network server. Each workstation can install a copy of the EJST. With these infrastructures set up, member users can login to this keystore through the Internet.

Second, one of the weaknesses on EJST is the time required for storing and retrieving entries on a keystore. Our tests did not reveal any problem on this. However, if the keystore is installed on a wide area network or on the Internet, where the keystore may be responsible for handling thousands of entries, it may not be able to afford the workload. In this case, the entries should be stored in a distributed database management system (DBMS). Modification to KCPM should be made so that it can communicate with the DBMS through the Java Database Connectivity (JDBC).

Finally, when importing certificates, users should, first, check if the certificates are invalid. To do that, users can consult a certificate revocation list (CRL), which is a list of certificates that have been revoked. This list can be obtained from a CA. Modification to the EJST should be made so that certificate revocation check can be done automatically when importing certificates.

REFERENCES

1. Yasin, Rutrell. "IETF Initiatives Gives Boost to Embattled SSL" Internet Week 6 July 1998: 1-2.

2. Yasin, Rutrell. "An Embattle standard Gets a Shot in the Arm." Internet Week 13 July 1998: 28-29.

3. "RSA Extends Lead in Java Security Race with BSAFE SSL-J 2.1 Software." Online. Internet. June 1999. Available: http://industry.java.sun.com.

4. Stallings, William. "Pretty Good Privacy." Byte July 1994: 193-195.

5. Levitt, Jason. "What is Public Key Infrastructure?" Information Week Jan. 2000: 82-83.

6. Levine, Daine. "Public Key Infrastructure Adds Security To E-Business." Information Week May 2000: 94-96.

7. Yasin, Rutrell. "PKI Crosses Enterprise Boundaries." Internet Week 1 May 2000: 1-2.

8. Radcliff, Deborah. "Digital Signatures." Computer World April 2000: 64.

9. Jim Kerstetter and Paul Korzeniowski. "Internet Privacy--and Piracy." PC Week 16 June 1997: 138.

10. Jim Kerstetter and Scot Petersen. "Web Server Security Spec Gaining Support." PC Week 28 June 1999: 1.

11. Kosiur, Dave. "Securing Internet VPNs." PC Week 25 Aug. 1997: 89-90.

12. Wirbel, Loring. "Push is on for Vistual-Private-Network Solutions." Electronic Engineering Times Mar. 1999: 20.

13. Ulfelder. Steve. "VPNX 101." Computer World 6 Mar. 2000: 80.

14. "Crunching Internet Security Codes." Science News 156.14 (1999): 221.

15. Ganesan, Srinivasa, and Madhusudan Sastry. "Time is Right for a Good, Secure 'Idea'." Electronic Engineering Times 23 Oct 1995: 66-68.

16. Kerstetter, Jim. "RSA Opens up S/MIME." PC Week 1 Sept. 1997: 8-9.

17. McQuilken, Barry. "Securing the Enterprise Network." Telecom Asia 8.6 (1997): 74-76.

18. Mantakos, Harry. "The Java OTP Calculator." Online. Internet. Available: http://www.cs.umd.edu

19. Brieva, Art. "Cover Your Assets - Fending off Hackers Requires a Mix of Firewall Technologies." Computer Shopper July 2000: 236-240.

20. Burr, Nazario, and Timothy Polk. "A Proposed Federal PKI Using X.509 V3 Certificates." Online. Internet.

21. "Puzzling Secrets: Cryptography." The Economist 7 Sept. 1996: 79-80.

22. Knudsen, Jonathan. Java Cryptography. California: O'Reilley & Associates, Inc., 1998.

23. Postoia, Marco, et al. Java 2 Network Security. New Jersey: Prentice Hall, 1999.

24. Oaks, Scott. Java Security. California: O'Reilley & Associates, Inc., 1999.

25. Petrie, C.S., and J.A. Connelly. "The Sampling of noise for random number generation." IEEE Circuits and Systems 6 (1999): 26-29.

26. Karras, D.A., and V. Zorkadis. "Overfitting in Multilayer Perceptrons as a Mechanism for (Pseudo) Random Number Generation in the Design of Secure Electronic Commerce Systems." IEEE Information Systems for Enhanced Public Safety and Security (2000): 345-349.

27. Horton, Ivor. Beginning C++, The Complete Language. United Kingdom: Wrox Press, 1998.

28. Hughes, Larry. Internet Security Techniques. Indiana: New Riders, 1995.

29. Deng, Lih-Yuan and Dennis Lin. "Random Number Generation for the New Century." The American Statistician May (2000): 145-150.

30. L'Ecuyer, Pierre. "Uniform Random Number Generators." IEEE Simulation Conference Proceedings Dec. (1998): 97-104.

31. Drew, Grady. Using SET for Secure Electronic Commerce. New Jersey: Prentice-Hall, Inc., 1998.

32. Adams, Carlisle, et al. "Which PKI (Public Key Infrastructure) is the Right One?" Proceedings of the 7th ACM Conference on Computer and Communications Security (2000): 98-101.

33. Boyarsky, Maurizio. "Public-key Cryptography and Password Protocols: The Multi-User Case." Proceedings of the 6th ACM Conference on Computer and Communications Security (1999): 63-72.

34. Kyas, Othmar. Internet Security Risk Analysis, Strategies and Firewalls. London, UK: International Thomson Computer Press, 1997.

35. Hoover, D.N. and Kausik, B.N. "Software Smart Card Via Cryptographic Camouflage." IEEE Security and Privacy Proceedings of the 1999 Symposium May (1999): 208 – 215.

36. Lee, Yung-Cheng and Laih, Chi-Sung. "On the Key Escrow System Without Key Exchange." Computers & Electrical Engineering July (1999): 279 – 280.

37. Schneier, Bruce. Applied Cryptography. New York: Wiley, 1996.

38. Horstmann, Cay and Cornell, Cary. Core Java Volume I-Fundamentals. California: Sun Microsystems, Inc., 1997.

39. Robinson, Matthew and Vorobiev, Pavel. Swing. Connecticut: Manning Publications, 2000.

40. Zukowski, John. Definitive Guide to Swing for Java 2, second edition. California: Apress, 2000.

41. Geary, David. Java 2 Mastering the JFC, third edition. California: Sun Microsystems, 1999.

42. Horstman, Cay and Cornell, Gary. Core Java 1.1, volume 1. New York: Prentice Hall, 1997.

43. Chan, Patrick, et al. The Java Class Libraries, second edition, volume 1. Massachusetts: Wesley, 1999.

44. Palmer, Grant. Java Programmer's Reference. Wrox Press, 2000.

45. "PKCS #5 v2.0: Password-Based Cryptography Standard." RSA Laboratories. Mar. 1999.

46. PKCS #7 v1.5: "Cryptographic Message Syntax Standard." RSA Laboratories. Nov. 1993.

47. PKCS #8 v1.2: Private-Key Information Syntax Standard." RSA Laboratories. Nov. 1993.

48. "PKCS #10 v1.7: Certificate Request Syntax Standard." RSA Laboratories. May 2000.

49. "Java Cryptography 1.2.1 API Specification & Reference." Online. Internet. Jun. 2000. Available: http://java.sun.com.

Appendix A. Acronyms

API:	Application Programming Interface
ASCII:	American Standard Code for Information Interchange
B2B:	Business-to-Business
B2C:	Business-to-Customer
CA:	Certificate Authority
CBC:	Cipher Block Chaining (mode)
CFB:	Cipher Feedback (mode)
CRL:	Certificate Revocation List
DBMS:	Database Management System
DES:	Digital Encryption Standard
DH:	Diffie-Hellman (key-exchange algorithm)
DN:	Distinguished Name (X.509)
DSA:	Digital Signature Algorithm
ECB:	Electronic Codebook (mode)
EJST:	Enhanced Java Security Tools
GUI:	Graphical User Interface
IDEA:	International Data Encryption Algorithm
IPES:	Improved Proposed Encryption Standard
ISO:	International Standards Organization
JAR:	Java Archive
JCA:	Java Cryptography Architecture
JCE:	Java Cryptography Extension
JDBC:	Java Database Connectivity
JDK:	Java Development Kit
JRE:	Java Run-time Environment
JVM:	Java Virtual Machine
KCPM:	Key-Certificate-Policy Manager
MAC:	Message Authentication Code
MD:	Message Digest
NIST:	National Institute of Standards and Technology
OFB:	Output Feedback (mode)
OTP:	One-time Password
PGP:	Pretty Good Privacy
PIN:	Personal Identification Number
PKI:	Public Key Infrastructure
PKCS:	Public-Key Cryptography Standard
PRNG:	Pseudo-random Number Generator
RCA:	Root Certificate Authority
RSA:	Rivest Shamir Adleman (public key algorithm)
SDK:	Standard Development Kit

SHA:	Secure Hash Algorithm
SSL:	Secure Socket Layer
TDES:	Triple Digital Encryption Standard (also know as DESede)
TRS:	Truly Random Seeders

Appendix B. Glossary

Base 64:	An Internet standard that can be used to print certificates. It rearranges the bits of the data stream in such a way that only the six least significant bits are used in every byte [23].
No encoding with: public key and signature	It is used to print information from certificates that can be read from human. This information includes: certificate creation date and type, the owner's and the issuer's X.500 DN, public key, and the signature.
No encoding with: fingerprints	It is used to print information from certificates that can be read from human. This information includes: certificate creation date and type, the owner's and the issuer's X.500 DN, MD5 fingerprint, and SHA fingerprint.
SHA1WithDSA:	Signature algorithm using SHA1 and DSA.
MD2WithRSA:	Signature algorithm using MD2 and RSA.
MD5WithRSA:	Signature algorithm using MD5 and RSA.
SHA1WithRSA:	Signature algorithm using SHA1 and RSA.
java.home:	JDK installation directory (i.e. c:\jdk1.3\jre on Windows)
user.home:	Operating system's installation directory (i.e. c:\windows on Windows)
passphrase:	A phrase that can be used as a password.
random key:	A string that contains random characters.
keystore:	A key database that is used to store key entries and certificate entries.
key entry:	Contains a private key and the associated certificate chain.
certificate entry:	Contains a single trusted certificate.
PBEWithMD5AndDES:	A password-based encryption algorithm. It uses DES as the symmetric cipher, cipher block chaining (CBC) mode as the

	cipher mode, PKCSPadding as the padding scheme, and MD5 as the hash function.
salt:	A random string that is concatenated with passwords before being operated on by the one-way function.
fingerprint:	The hash value of a set of data.

VITA

Jp-Kin Anthony Wong

Candidate for the Degree of

Master of Science

Thesis: ENHANCED JAVA SECURITY TOOLS

Major Field: Computer Science

Biographical:

Education: Received Associate of Arts in Liberal Arts from Maui Community College, Kahului, Hawaii. Received Bachelor of Science degree in Computer Science from University of Central Oklahoma, Edmond, Oklahoma. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December, 2001.

Experience: Employed by the University of Central Oklahoma, College of Business as a lab monitor; University of Central Oklahoma, College of Business, 1996 - 1997. Employed by the Pardon and Parole Board as an application developer; Oklahoma Pardon and Parole Board, Data Processing Department, 1998 - 1999. Employed by the Department of Correction as an application specialist II; Oklahoma Department of Correction, Data Processing Department, 1999. Employed by Oklahoma State University, Department of Computer Science as a research assistant; Oklahoma State University, Department of Computer Science, 1999 - 2001. Employed by Oklahoma State University, Department of Computer Science as a teaching assistant; Oklahoma State University, Department of Computer Science, 2000 - 2001.

Professional Memberships: Student in Free Enterprise, Web Master's Club.