

Secure Web Development Teaching Modules¹

Introduction to Java Security

Contents

1	Concepts.....	2
1.1	Basic Terms.....	3
1.2	Java Security Framework.....	4
1.3	Key Management	5
2	Lab Objectives	6
3	Lab Setup.....	6
4	Lab Guide	6
4.1	Reviewing Java Security Framework.....	6
4.2	Creating Public/Private Keys and Digital Certificates	8
4.3	Ensuring Data Confidentiality with Cryptography	10
4.4	Securing File Exchange with Java Security Utilities	11
4.5	Granting Special Rights to Applets Based on Code Location.....	12
4.6	Granting Special Rights to Applets Based on Code Signing	20
4.7	Creating a Certificate Chain to Implement a Trust Chain.....	26
4.8	Protecting Your Computer from Insecure Java Applications	31
4.9	Securing File Exchange with Java Security API and Newly Created Keys.....	32
4.9.1	Java command-line arguments	33
4.9.2	Generating public/private keys.....	33
4.9.3	Generating a digital signature.....	34
4.10	Securing File Exchange with Java Security API and Keys in Files.....	35
4.10.1	Importing a private key from a file.....	35
4.11	Securing File Exchange with Java Security API and Keys in a Keystore	36
4.11.1	Importing a private key and certificate pair from a keystore.....	36
4.11.2	Importing a public key from a digital certificate file.....	37
5	Review Questions	37
6	Appendix.....	39
6.1	File “GetProperties.java”	39
6.2	File “WriteFile.java”	39
6.3	File “appletWrite.html”	40
6.4	File “appletWrite2.html”	40
6.5	File “my.java.policy”	40
6.6	File “GenerateSignature.java”	40
6.7	File “VerifySignature.java”	41
6.8	File “GenerateKeys.java”	42

¹ Copyright© 2009-2011 Li-Chiou Chen (lchen@pace.edu) & Lixin Tao (ltao@pace.edu), Pace University. This document is based upon work supported by the National Science Foundation’s Course Curriculum, and Laboratory Improvement (CCLI) program under Grant No. 0837549. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation. A copy of the license is available at <http://www.gnu.org/copyleft/fdl.html>.

6.9	File “GenerateSignature2.java”	44
6.10	File “GenerateSignature3.java”	44

Prerequisite: Completion of SWEET Security Lab Series module “Introduction to Cryptography”.

1 Concepts

Java is a popular programming language for server-side computing for two main reasons. First it is among the first languages that support the more efficient light-weight threads, instead of the heavy-weight processes, to support multi-tasking and multi-processor computing which have important impact on server performance. Second as an interpreted language it could check each bytecode instruction for security vulnerabilities just before the instruction is scheduled to run. At the same time Java is also popular on client platforms. Java JRE plug-ins enable web browsers to run Java applets to extend browser functionality, and standalone Java applications provide GUI-rich services running directly on the operating systems. This tutorial introduces the concepts and tools for supporting Java security.

Since the Java applets are downloaded from the web and not explicitly activated by the users, they introduce severe security concerns. By default all applets always run under the constraints of a Java security manager. The example constraints for applets include

- Applets cannot access user computer resources like keyboard and files.
- Applets can only communicate with the web site from which they were downloaded.

In this tutorial you will learn how to assign special rights to secure applets so they could provide more functionality.

On the other hand, by default Java applications are usually activated by the users and thus considered safe and not constrained by the Java security manager. But if an application is downloaded from the web, we would not be sure of its security. In this tutorial you will learn how to run such stand-alone applications under the supervision of the Java security manager so you could control which resources of yours could be used by the applications.

In the tutorial “Introduction to Cryptography” you learned how to use GPG (GNU version of PGP) to generate public/secret keys and digital signatures to secure data communications. Java has utilities to accomplish the similar tasks. Java also has application programming interfaces (API) to secure applications. In this tutorial you will learn how to secure your applets and applications with Java security utilities, and review sample Java code to see how selected security tasks could be accomplished with the Java APIs.

Question 1: Why is Java popular for server-side or enterprise computing?

Question 2: What are the main differences between Java applications and Java applets?

Question 3: Why the Java platform normally trusts applications but not applets?

Question 4: What are the differences between Java utilities and Java API?

1.1 Basic Terms

Encryption/decryption algorithms are the basic tools for computer security. To encrypt a plain document, we need a key, which is a small piece of fixed-length data. The encryption algorithm reads the plain document and a key and produces the encrypted document. The decryption algorithm reads the encrypted document and a corresponding key to regenerate the original plain document. There are two categories of encryption/decryption algorithms. The symmetric or secret key algorithms use the same key for encryption and decryption. These algorithms are more efficient for encrypting/decrypting large volume of data. The public key algorithms use a pair of public and private keys: if a document is encrypted by one of the two keys, the other key can be used to decrypt the document. These algorithms are mainly for secret key exchange, identity authentication and data validation.

For a person to distribute a document secretly, he could first generate a pair of private/public keys as his own identity (he could reuse these keys), use the private key to encrypt the document, and send the encrypted document and the public key to the receiver. The receiver then can use the public key to decrypt the document, and be assured that the document is from the sender (identity authentication) and the document has not been modified (data validation). But here we assume that the public key itself has been distributed in a secure way, which is normally implemented with digital signatures and certificates explained below.

Sometimes the document contents are public, and we just need to assure the document receiver that the document is originated from the right sender and it has not been modified along the way. In this case you could use digital signatures to achieve sender identity authentication and data validation without the time-consuming encryption/decryption process for the document itself. The document is first transformed by a hash function (typically MD5 or SHA1) into a fixed-length short sequence of bytes, called a fingerprint, so that each byte of the fingerprint depends on many characters of the document and any change to the document would change the fingerprint with high probability. The fingerprint is then encrypted with the sender's private key into a fixed-length digital signature. As long as the receiver gets a copy of the plain document, its digital signature, and the sender's public key, the sender could reproduce the fingerprint from the digital signature and the sender's public key, generate his own copy of fingerprint from the plain document, and compare the two fingerprints. If they are the same, the document was originated from the sender and it has not been compromised along the way. Otherwise the document should not be trusted.

Now we can consider the earlier problem of how we could securely distribute the public keys, a critical step in identity authentication and data validation. A digital certificate is used to certify the identity and public key of a digital signer, who is a person or a company and needs to assure other people of the authenticity of his documents or applications). A certificate is a small record containing the digital signer's public key and identity information (work unit, company, address), and the digital signature of the public key and identity information generated with a private key of the certifier, the person or company that certifies the authenticity of information in this certificate. If the certifier is the digital signer himself, the certificate is self-certified and it does not authenticate the information in the certificate. If the certifier is another person or company trusted by a user of the certificate, then the user's trust to the certifier can now be used to authenticate the information in the certificate. A few certificate authority (CA) companies, including VeriSign and GTE, are set up and supposed to be trusted by the public, and their self-certified certificates are distributed to user computers as trusted certificates either by software (OS or application) installation, or by the user's agreement. To distribute his public key to the public, a signer generates a self-certified certificate containing his public key and identity information, sends it with payment to one of the CAs to apply for certifying the certificate by the CA for a period of time (six month or longer). The CA would verify the signer's information, replace the self-certified certificate with the one certified by the CA, and send it back to the signer for distribution. The trust chain described here

could be extended: If A certifies B, B certifies C, and C certifies D; and the user trusts A, then the user can trust B, C and D too.

From the above discussion you can see that the key/certificate management is critical to the security of a computer.

Question 5: What is identity authentication?

Question 6: What is data validation?

Question 7: What is the most important task in computer security based on cryptography?

Question 8: What is the difference between a fingerprint and a signature of a document?

Question 9: What is the difference between a public key and its digital certificate?

1.2 Java Security Framework

A computer may have multiple users. Each user has its own home folder. Let us assume that a user John has login name “john”. If the computer runs Windows, John has “C:\users\john” as his home folder, which has “file://C:/users/john” as its URL. If the computer runs Linux, John has “/home/john” as his home folder, which has “file:///home/john” as its URL.

When you install Java JDK on a Linux computer, the folder holding the JDK installation is called the Java home (we represent it with [Java home] in this document), and the folder “jre” nested inside the Java home is called the Java JRE (Java runtime environment) home (we represent it with [JRE home] in this document). If you install Java JDK on a Windows computer, by default your Java home folder is “C:\Program Files\Java\jdk1.x_y” (where x and y specify version numbers); if you also see a folder “C:\Program Files\Java\jre#” (where # is a number), your Java JRE home folder is “C:\Program Files\Java\jre#” (replacing # with the actual digit). Most Java utilities for security management are in folder “[Java home]/bin”, so it is important to include this “bin” folder in the operating system “PATH” value. Most Java security policy related files are in folder “[JRE home]/lib/security”.

The *security manager* is the main mechanism for Java to assign access rights to Java programs. All applets always run under the control of the Java security manager and there is no way to opt out. A Java application, Java program that runs outside of a web browser, by default doesn’t run under the control of a security manager so it has full rights on the computer. If you don’t completely trust this application, you should run the application under the control of the Java security manager with a Java command-line switch as shown below:

```
Java -Djava.security.manager [Java Class Name]
```

Following the principle of separating policies from the mechanisms, the Java security manager does not hard code access rights to Java programs. When an applet or a application using Java security manager starts, Java security manager first reads file “[JRE home]/lib/security/java.security” to find where to load the java security policy files in the order specified by lines similar to

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

where the value of “policy.url.1” specifies the default security policy in file “[JRE home]/lib/security/java.policy” (SUN chose a bad name “java.home” to indicate what we call JRE home) for all users; and the value of “policy.url.2” specifies that the security policy in file “[user home]/.java.policy” (don’t forget the leading period); if user name is “john”, “[user home]” is “home/john” or “~” on a Linux computer, and “C:\users\john” on a Windows computer) for a particular user. You could also insert lines like

```
policy.url.3=file:${user.home}/john.policy
```

to add the extra Java policy files with decreasing priority. A policy file loaded later could override policies specified in a policy file loaded earlier.

While you are not supposed to modify file “[JRE home]/lib/security/java.policy”, you could freely modify file “[user home]/.java.policy” to modify existing policies and add new policies. The contents of Java security policy files have strict syntax requirements so it is easier to use the graphic user interface of Java utility “policytool” to review, edit, insert and delete policies.

Question 10: What is the home folder of *user* on the ubuntu10 VM? How to write it in URL format?

Question 11: What is your home folder on your Windows PC if your login name is “john”? How to write it in URL format?

Question 12: What is the difference between Java JRE and Java JDK?

Question 13: Which folder holds the most important Java security files?

Question 14: Which file specifies the location of your Java security policy files?

Question 15: Which is your default personal Java security policy file?

Question 16: Which Java utility is for helping you create Java security policy files?

Question 17: What is the Java security manager? When is it used?

1.3 Key Management

Java maintains public/private key pairs and digital certificates in keystore files with any file names and at any file system locations. A computer can have multiple keystore files. You can use Java utility “keytool” to create a new public/private key pair and insert it into an existing keystore file, or create a new keystore file first if necessary. A keystore file is protected by a keystore password. Each public/private key pair in the keystore is also protected by its own key password, which could be the same as the keystore password. When you create a new public/private key pair, you assign an alias or nickname to the pair for you later referring to the pair.

For more information on Java security, please visit the “Java SE Security” page at <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html>.

Question 18: Where are public/private keys normally stored?

Question 19: Which Java utility helps you maintain keystores?

2 Lab Objectives

In this lab you will

1. Learn and practice how to review your current Java policies;
2. Learn and practice how to create a Java applet, observe how it is limited in accessing system resources, and create a policy to enable it to access some system resources;
3. Learn and practice how to run a Java application with Java security manager, and create a policy to enable it to access some system resources;
4. Learn and practice how to use a Java utility to create public/private key pairs and digital certificates;

3 Lab Setup

You will use the *ubuntu10* VM for this module's labs. Launch the Ubuntu VM with username "user" and password 123456.

4 Lab Guide

4.1 Reviewing Java Security Framework

1. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
2. Run "cd ~/JavaSecurityLab" to change the work folder to "~/JavaSecurityLab".
3. Run "gedit GetProperties.java" to review the source code of file "GetProperties.java" as shown below:

```
class GetProperties {
    public static void main(String[] args) {
        String s;
        try {
            s = System.getProperty("os.name", "not specified");
            System.out.println(" Your operating system is: " + s);
            s = System.getProperty("java.version", "not specified");
            System.out.println(" Your Java version is: " + s);
            s = System.getProperty("user.home", "not specified");
            System.out.println(" Your user home directory is: " + s);
            s = System.getProperty("java.home", "not specified");
            System.out.println(" Your JRE installation directory is: " + s);
            s = System.getProperty("java.ext.dirs", "not specified");
            System.out.println(" Your Java extension directories are: " + s);
        } catch (Exception e) {
            System.err.println("Caught exception: " + e);
        }
    }
}
```

“os.name”, “java.version”, “user.home”, “java.home” and “java.ext.dirs” are Java property names for OS name, Java version, user home folder, Java JRE home folder, and Java extension folders. Method “String System.getProperty(String name, String defaultValue)” returns the value of the specified Java property name, or the default value if the property name is not defined.

4. Run “javac GetProperties.java” to compile the source code into bytecode file “GetProperties.class”.
5. Run “java GetProperties” to execute file “GetProperties.class”. You will see the following printout:

```
Your operating system is: Linux
Your Java version is: 1.6.0_16
Your user home directory is: /home/user
Your JRE installation directory is: /home/user/tools/jdk1.6.0_16/jre
Your Java extension directories are:
    /home/user/tools/jdk1.6.0_16/jre/lib/ext:/usr/java/packages/lib/ext
```

6. Click on menu item “Places|Home Folder” to launch the file explorer, and click to drill down to folder “/home/user/tools/jdk1.6.0_16/jre/lib/security”.
7. Right-click on file “java.security” and choose “Open with gedit” to review the contents of the file. Scroll to find the following two lines.

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

8. Right-click on file “java.policy” and choose “Open with gedit” to review the contents of the file. The following first “grant” statement

```
grant codeBase "file:${java.ext.dirs}/*" {
    permission java.security.AllPermission;
};
```

grants all permission to all Java Jar files in Java extension folders. The second “grant” statement enables all Java programs to read selected Java properties.

Java has two folders for extending its functions. Folder “[JRE home]/lib/ext” is for holding Jar files (zipped Java classes) used when these classes are not found in the standard Java library. The folder “[JRE home]/lib/endorsed” is for holding Jar files overriding the same named classes in the standard Java library.

9. Run “cd ~” to change work folder to be the user home folder. Run “ls -alg” to verify that there is no file “java.policy” in the user home folder yet.

Question 20: What are the two major vulnerabilities of Java security framework?

Question 21: How should you resolve the vulnerabilities of Java security framework?

4.2 Creating Public/Private Keys and Digital Certificates

1. Launch the Ubuntu VM with username “user” and password 12345678.
2. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.
3. Run “cd ~/JavaSecurityLab” to change the work folder to “~/JavaSecurityLab”.
4. Run Java utility “keytool” as in the following line (on the same line, no line break) to create a pair of public/private keys, assign alias “PaceKey” to this pair of keys, assign key password “Seidenberg” to this pair of keys, create a new keystore file “PaceKeystore” and set its keystore password to be “Pace” if there is no such a file in the current folder yet, access keystore file “PaceKeystore” with keystore password “PaceUniversity”, and insert the new key pair in the keystore.

```
keytool -genkey -alias PaceKey -keypass Seidenberg -keystore  
PaceKeystore -storepass PaceUniversity
```

When prompted, enter “John Smith” as first and last names, “Seidenberg” for organizational unit, “Pace University” as organization, “New York” as city, “NY” as state, “US” as name of two-letter country code, and “y” to confirm your information. The information that you just entered interactively is called your *distinguished-name* information. Since you don’t have a keystore file “PaceKeystore” in the current folder, this file is created in the current folder. This command also stored in this keystore a pair of new public/private keys and a self-certified digital certificate that includes the public key and the distinguished-name information. This certificate will be valid for 90 days, the default validity period if you don’t specify a *-validity* option. The certificate is associated with the public/private key pair in a keystore entry referred to by the alias “PaceKey”.

```
user@ubuntu:~/JavaSecurityLab$ keytool -genkey -alias PaceKey -keypass  
Seidenberg -keystore PaceKeystore -storepass PaceUniversity  
What is your first and last name?  
[Unknown]: John Smith  
What is the name of your organizational unit?  
[Unknown]: Seidenberg  
What is the name of your organization?  
[Unknown]: Pace University  
What is the name of your City or Locality?  
[Unknown]: New York  
What is the name of your State or Province?  
[Unknown]: NY  
What is the two-letter country code for this unit?  
[Unknown]: US  
Is CN=John Smith, OU=Seidenberg, O=Pace University, L=New York, ST=NY,  
C=US correct?  
[no]: y
```

If you don’t specify the keystore file name, “keytool” will use the default file “[user home]/.keystore”.

5. Run the following command to export a copy of your newly generated public key, with alias “PaceKey”, in a self-certifying digital certificate file “PaceKey.cer”.

```
keytool -export -keystore PaceKeystore -alias PaceKey -file PaceKey.cer
```

“keytool” will ask you for the keystore password “PaceUniversity”, and then generate the new certificate file “PaceKey.cer”.

```
user@ubuntu:~/JavaSecurityLab$ keytool -export -keystore PaceKeystore
-alias PaceKey -file PaceKey.cer
Enter keystore password: PaceUniversity
Certificate stored in file <PaceKeys.cer>
```

6. Run “keytool -printcert -file PaceKey.cer” to review the contents of the certificate in file “PaceKey.cer”, and “keytool” will print out the following information:

```
user@ubuntu:~/JavaSecurityLab$ keytool -printcert -file PaceKey.cer
Owner: CN=John Smith, OU=Seidenberg, O=Pace University, L=New York, ST=NY,
C=US
Issuer: CN=John Smith, OU=Seidenberg, O=Pace University, L=New York,
ST=NY, C=US
Serial number: 4ca4dd45
Valid from: Thu Sep 30 11:56:05 PDT 2010 until: Wed Dec 29 10:56:05 PST
2010
Certificate fingerprints:
  MD5: CA:09:3C:EC:12:D9:5D:20:E8:1E:6A:FB:88:CE:B2:18
  SHA1: F7:CD:4B:18:66:55:C8:2B:BB:9E:AD:92:A8:DF:54:9A:F7:96:93:E7
Signature algorithm name: SHA1withDSA
Version: 3
```

Note the certificate fingerprints in MD5 and SHA1 formats. If you send this certificate to a friend, you could both run this command to find the fingerprints of this certificate, and compare them to see whether the certificate (signed public key) has been compromised in transit.

7. If you were doing the real work, you may now contact one of the CAs, send it your self-signed certificate file “PaceKey.cer” with your payment, and the CA will verify your distinguished-name information and send back to you your digital certificate signed by the CA. The CA certified certificate is the one that you are supposed to distribute to your partners. In this tutorial we skip this step and just distribute the self-certified certificate in file “PaceKey.cer”.
8. In the real situation, your digital certificate should be distributed to your partners, and imported to your partners’ keystores. In this tutorial you will also act as your partner. You will import the newly generated digital certificate in file “PaceKey.cer” into a new keystore file named “receiverKeystore” in the current folder with the following command

```
keytool -import -alias Pace -file PaceKey.cer -keystore
receiverKeystore
```

Upon being asked for keystore password, enter “123456”. When being asked whether you trust this certificate, respond with “y” for yes.

```
user@ubuntu:~/JavaSecurityLab$ keytool -import -alias Pace -file
PaceKey.cer -keystore receiverKeystore
Enter keystore password: 123456
Re-enter new password: 123456
Owner: CN=John Smith, OU=Seidenberg, O=Pace University, L=New York, ST=NY,
C=US
Issuer: CN=John Smith, OU=Seidenberg, O=Pace University, L=New York,
ST=NY, C=US
Serial number: 4ca4dd45
```

```
Valid from: Thu Sep 30 11:56:05 PDT 2010 until: Wed Dec 29 10:56:05 PST 2010
Certificate fingerprints:
  MD5: CA:09:3C:EC:12:D9:5D:20:E8:1E:6A:FB:88:CE:B2:18
  SHA1: F7:CD:4B:18:66:55:C8:2B:BB:9E:AD:92:A8:DF:54:9A:F7:96:93:E7
  Signature algorithm name: SHA1withDSA
  Version: 3
Trust this certificate? [no]: y
Certificate was added to keystore
```

Question 22: What is the more secure way to run *keytool* to generate the public/private keys?

4.3 Ensuring Data Confidentiality with Cryptography

Protecting confidential data is a very important task of computer security. Most of the labs in this tutorial focus on identity authentication and data validation and they don't encode/decode the data files. In this lab you will try out Prof. Lixin Tao's implementation of the *Simplified DES* algorithm described in file "JavaSecurityLab/cipher-des/C-SDES.pdf". This implementation aims at helping students understand the algorithm, which is not the objective of this lab. With the hands-on experience with applying the DES cipher you could integrate it into the latter labs to add the data confidentiality component into them. DES is a very fundamental algorithm in cryptography. If you are taking a course in which cryptography is a topic, make sure your instructor explains this algorithm and you understand its design and my implementation. I have used extra comments and clear code design to help you read my DES implementation. But not all students working on this lab needs to read my source code.

1. Launch the Ubuntu VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab/cipher-des" to change work folder to "~/JavaSecurityLab/cipher-des".
4. Run "javac S_DES_File.java" to compile all the four Java source code files into their corresponding bytecode files.
5. Run "java S_DES_File" to find out how to run this program.

```
usage: java S_DES_File inputFile outputFile key-bits [decode]
```

The program takes one input file and one output file. For encoding with DES, the input file is the data file, and the output file is its encoded version. For decoding, the input file is the encoded file, and the output file is the decoded data file. In either case you need to provide "key-bits", the secret key in form of 1-10 binary bits (the professional implementation will use much longer keys; short keys allow you to trace the DES algorithm if you turn on the debugging mode of the program). Make sure that you use the same key to encode and decode a file. The "decode" switch is needed only when you decode a file.

6. Run "java S_DES_File GettysburgAddress.txt secret 1010101010" to use the DES algorithm to encode file "GettysburgAddress.txt" into a binary file "secret" with the secret key 1010101010.
7. Run "more secret" to confirm that you could not figure out the contents of file "secret".
8. Run "java S_DES_File secret GettysburgAddress2.txt 1010101010 decode" to decode file "secret" into plain file "GettysburgAddress2.txt" with the same secret key.
9. Run "more GettysburgAddress2.txt" to confirm that you have recovered the original file contents. Therefore your encode/decode process works correctly.

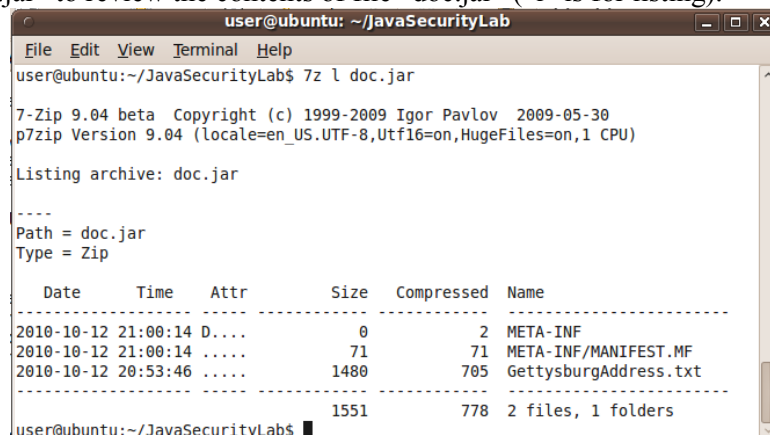
You can use this program to encode/decode any file of any size no matter it is a binary file or a text file.

4.4 Securing File Exchange with Java Security Utilities

In this lab you study how to use the public/private keys and the digital certificate in the last lab to securely send a document. The receiver of the document should be able to verify that the document was from the right person (identity authentication), and the document has not been modified in transit (data validation).

First let us recall what you have done in your second lab in this tutorial. You have created a pair of public/private keys in keystore file “~/JavaSecurityLab/PaceKeystore”, assigned alias “PaceKey” to this pair of keys, exported the self-signed digital certificate for the public key, imported the digital certificate for the public key into another keystore file “~/JavaSecurityLab/receiverKeystore”, and assigned alias “Pace” to this certificate. In this lab the imaginary document sender will use the private key with alias “PaceKey” in keystore file “~/JavaSecurityLab/PaceKeystore” to sign the Jar file of a sample document, and the imaginary document receiver will perform identity authentication and data validation for the received document using the public key with alias “Pace” in keystore file “~/JavaSecurityLab/receiverKeystore”.

10. Launch the Ubuntu VM with username “user” and password 123456.
11. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.
12. Run “cd ~/JavaSecurityLab” to change work folder to “~/JavaSecurityLab”.
13. Run “more GettysburgAddress.txt” to review its contents. You use file “GettysburgAddress.txt” as example of a secret document for file exchange.
14. Run “jar cvf doc.jar GettysburgAddress.txt” to generate a Jar file “doc.jar” for transmission.
15. Run “7z l doc.jar” to review the contents of file “doc.jar” (“l” is for listing).



```
user@ubuntu: ~/JavaSecurityLab
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab$ 7z l doc.jar
7-Zip 9.04 beta Copyright (c) 1999-2009 Igor Pavlov 2009-05-30
p7zip Version 9.04 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,1 CPU)

Listing archive: doc.jar
-----
Path = doc.jar
Type = Zip

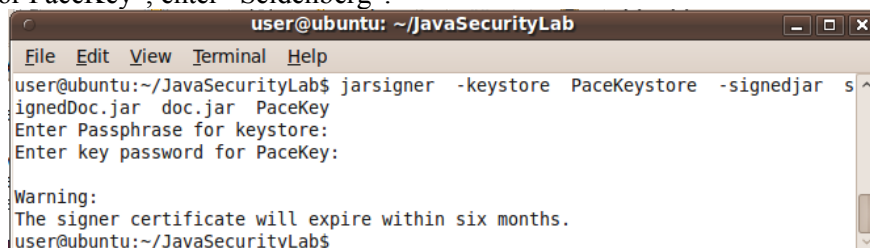
  Date       Time       Attr      Size   Compressed  Name
-----
2010-10-12  21:00:14  D....        0         2  META-INF
2010-10-12  21:00:14  ....         71        71  META-INF/MANIFEST.MF
2010-10-12  20:53:46  ....      1480       705  GettysburgAddress.txt
-----
                          1551       778  2 files, 1 folders

user@ubuntu:~/JavaSecurityLab$
```

16. Now run the following command to sign file “doc.jar” with the private key with alias “PaceKey” in keystore file “PaceKeystore”:

```
jarsigner -keystore PaceKeystore -signedjar signedDoc.jar doc.jar PaceKey
```

When asked for “passphrase for keystore”, enter “PaceUniversity”. When asked for “key password for PaceKey”, enter “Seidenberg”.

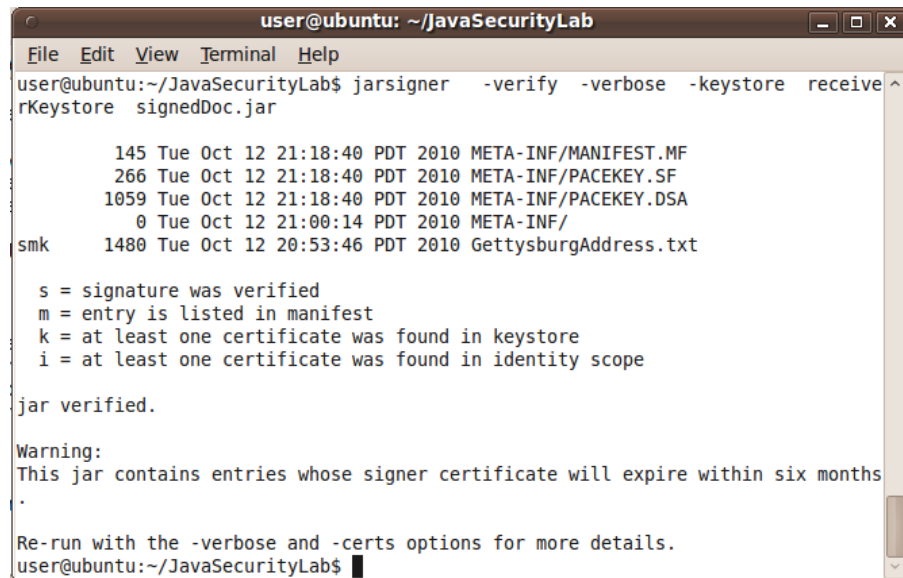


```
user@ubuntu: ~/JavaSecurityLab
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab$ jarsigner -keystore PaceKeystore -signedjar s
signedDoc.jar doc.jar PaceKey
Enter Passphrase for keystore:
Enter key password for PaceKey:

Warning:
The signer certificate will expire within six months.
user@ubuntu:~/JavaSecurityLab$
```

17. Now you will function as the document receiver. You just received file “signedDoc.jar”, and you have the digital certificate for the document signer in your keystore file “receiverKeystore” (you created them in the previous lab). Run the following command to authenticate the signer and make sure the Jar file has not been tampered with:

```
jarsigner -verify -verbose -keystore receiverKeystore signedDoc.jar
```



```
user@ubuntu: ~/JavaSecurityLab
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab$ jarsigner -verify -verbose -keystore receiverKeystore signedDoc.jar
145 Tue Oct 12 21:18:40 PDT 2010 META-INF/MANIFEST.MF
266 Tue Oct 12 21:18:40 PDT 2010 META-INF/PACEKEY.SF
1059 Tue Oct 12 21:18:40 PDT 2010 META-INF/PACEKEY.DSA
0 Tue Oct 12 21:00:14 PDT 2010 META-INF/
smk 1480 Tue Oct 12 20:53:46 PDT 2010 GettysburgAddress.txt

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.

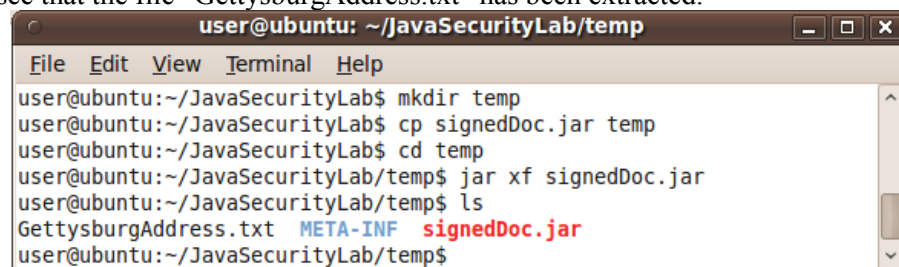
Warning:
This jar contains entries whose signer certificate will expire within six months
.

Re-run with the -verbose and -certs options for more details.
user@ubuntu:~/JavaSecurityLab$
```

The message shows that the Jar file was signed by the right person and the jar file has not been tampered with.

18. Run the following commands to retrieve file “GettysburgAddress.txt” from file “signedDoc.jar”
- “mkdir temp” to create a new folder “temp”;
 - “cp signedDoc.jar temp” to copy file “signedDoc.jar” into folder temp;
 - “cd temp” to change work folder to “temp”;
 - “jar xf signedDoc.jar” to extract the contents of file “signedDoc.jar”;
 - “ls” to list the files in folder “temp”.

You will see that the file “GettysburgAddress.txt” has been extracted.



```
user@ubuntu: ~/JavaSecurityLab/temp
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab$ mkdir temp
user@ubuntu:~/JavaSecurityLab$ cp signedDoc.jar temp
user@ubuntu:~/JavaSecurityLab$ cd temp
user@ubuntu:~/JavaSecurityLab/temp$ jar xf signedDoc.jar
user@ubuntu:~/JavaSecurityLab/temp$ ls
GettysburgAddress.txt META-INF signedDoc.jar
user@ubuntu:~/JavaSecurityLab/temp$
```

This concludes your lab for securely exchanging files.

4.5 Granting Special Rights to Applets Based on Code Location

1. Launch the Ubuntu VM with username “user” and password 123456.
2. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.

3. Run “cd ~/JavaSecurityLab/applet1” to change work folder to “~/JavaSecurityLab/applet1”.
4. Run “gedit WriteFile.java” to review the contents of the applet source file.

```
import java.awt.*;
import java.applet.*;
import java.io.*;

public class WriteFile extends Applet {
    public void paint(Graphics g) {
        try {
            String fileName = System.getProperty("user.home") +
                System.getProperty("file.separator") + //    / or \
                "data.txt";
            File f = new File(fileName);
            PrintWriter output = new PrintWriter(new FileWriter(f), true); // auto-flush
            output.println(new java.util.Date() + ": Pace University Java Tutorial");
            g.drawString("Data were successfully written to file \"" + fileName +
                "\"", 10, 10);
        }
        catch (SecurityException e) {
            g.drawString("WriteFile: security exception is caught", 10, 10);
            e.printStackTrace(); // print error messages to terminal window
                                // for debugging
        }
        catch (IOException ioe) {
            g.drawString("WriteFile: i/o exception is caught", 10, 10);
        }
    }
}
```

An applet is a Java program that is embedded in an HTML file and run inside a web browser. An applet class always extends the base class “Applet”. Its method “paint(Graphics g)” executes every time the web page containing it is reloaded. This applet first opens the file “[user home]/data.txt” (creating the file first if there is no such a file yet), and replaces its contents with the current time followed by “: Pace University Java Tutorial”. If for some reason the file cannot be opened or write fails, an exception will be thrown and displayed. Method call “System.getProperty(“file.separator”)” finds out whether the file separator in a file path is “/” or “\” so this applet could run on either Windows or Linux. Method call “g.drawString(“string”, 10, 10)” displays “string” in the applet area of the web browser 10 pixels to the right and 10 pixels down relative to the origin of the applet area specified with the “width” and “height” attributes of the <applet> element in an HTML file.

5. Shutdown the gedit editor.
6. Run “javac WriteFile.java” to compile the source code into bytecode file “WriteFile.class”.
7. Run “gedit appletWrite.html” to review the contents of file “appletWrite.html”.

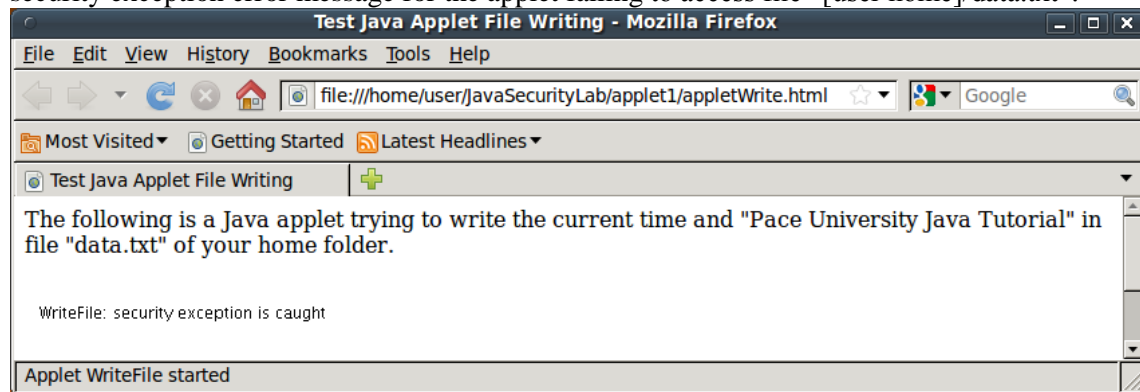
```
<html>
<head>
<title>Test Java Applet File Writing</title>
</head>

<body>
<p>The following is a Java applet trying to write the current time and
"Pace University Java Tutorial" in file "data.txt" of your home
folder.</p>
<br/>
<applet code="WriteFile.class" width=420 height=100>
</applet>
```

```
</body>
</html>
```

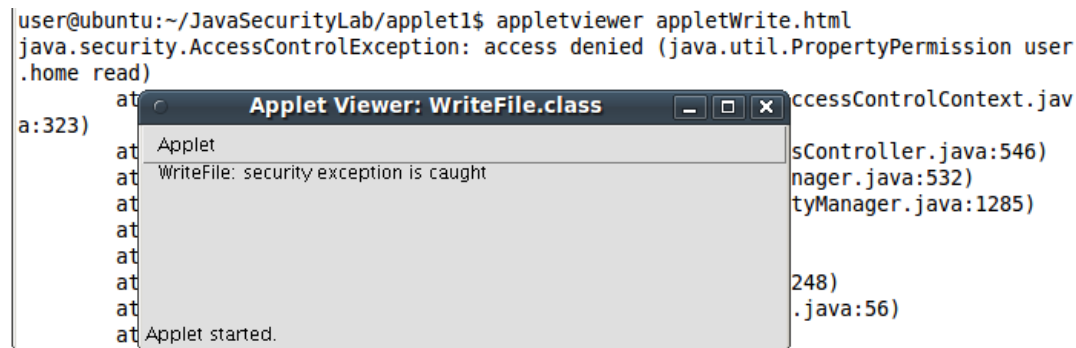
An applet is embedded in an HTML file with the “applet” tag. The applet element can use attribute “code” to specify the applet bytecode file path relative to the folder containing the current HTML file, and the “width” and “height” attributes to specify the display area of the applet in a web browser.

8. Shutdown the gedit editor.
9. Double-click on file “appletWrite.html” to display it in a web browser, and you will see the security exception error message for the applet failing to access file “[user home]/data.txt”:

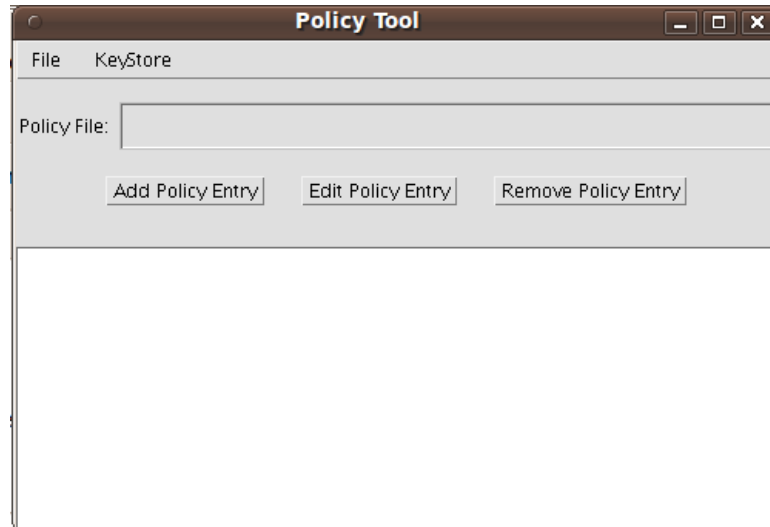


The applets always run under the Java security manager, which disables applets to access local files and many other resources. Since you didn't launch the web page with a terminal window, you cannot read the exception message printed by the “e.printStackTrace();” statement of the applet.

10. Now you will launch the web page with Java utility “appletviewer” so you could read more information about exceptions. Shutdown the web browser. Run “appletviewer appletWrite.html” to see the following applet window running the applet:

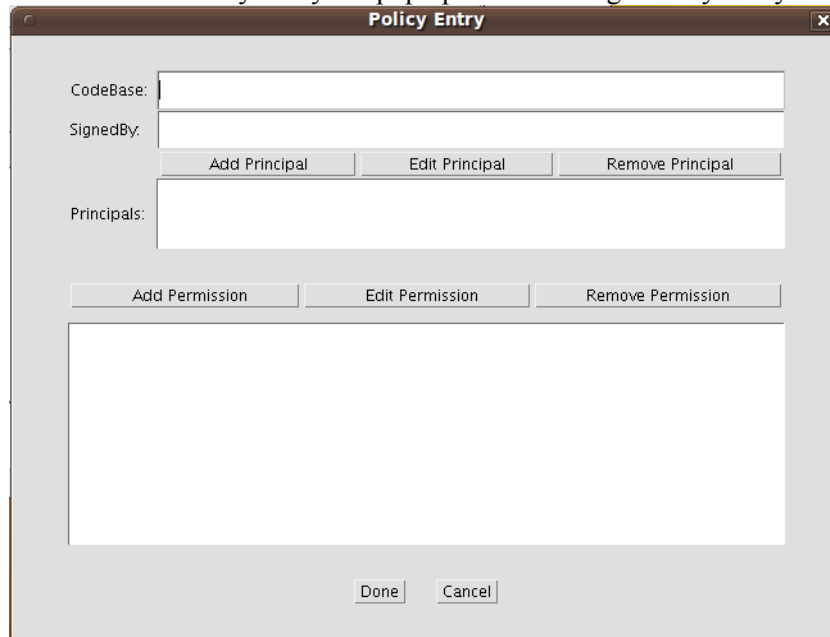


11. From the terminal window you could see a long list of error messages. You should focus on the first few lines, which told us that the applet has problem in accessing Java property “user.home”. Actually the exception only pointed out the first security problem. The second one is that the applet could not access local file “[user home]/data.txt”.
12. Now you will grant special rights for this applet to complete its task. Run Java utility “policytool” to launch the following Policy Tool window.

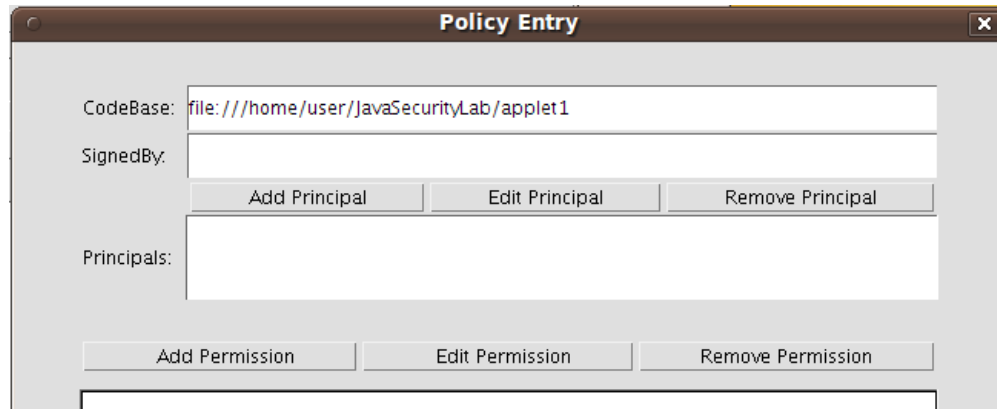


If you see contents in the “Policy File” textbox, it just means you already have a Java policy file for your account, and you can still continue in this lab.

13. Click on the button “Add Policy Entry” to pop up the following “Policy Entry” window.



14. You can use the CodeBase textbox to specify special rights to all Java code in a folder (CodeBase, specified with an absolute path) or its subfolders (terminating the CodeBase path with “/-”), or use the SignedBy textbox to specify special rights to all Java code signed by a particular private key where the SignedBy textbox holds the alias for the corresponding digital certificate stored in a keystore, or use both of the CodeBase and the SignedBy textboxes to specify special rights only to those Java code signed by a specific private key and located in a specific folder or its subfolders. In this exercise you only use the CodeBase textbox.
15. In the CodeBase textbox type “file:///home/user/JavaSecurityLab/applet1”, the URL for folder “~/JavaSecurityLab/applet1”. This will allow you to specify rights for all code in this folder. (If you add “/-” to the end of the URL, leading to “file:///home/user/JavaSecurityLab/applet1/-”, code in folder “applet1” or its subfolders have the same special rights.)



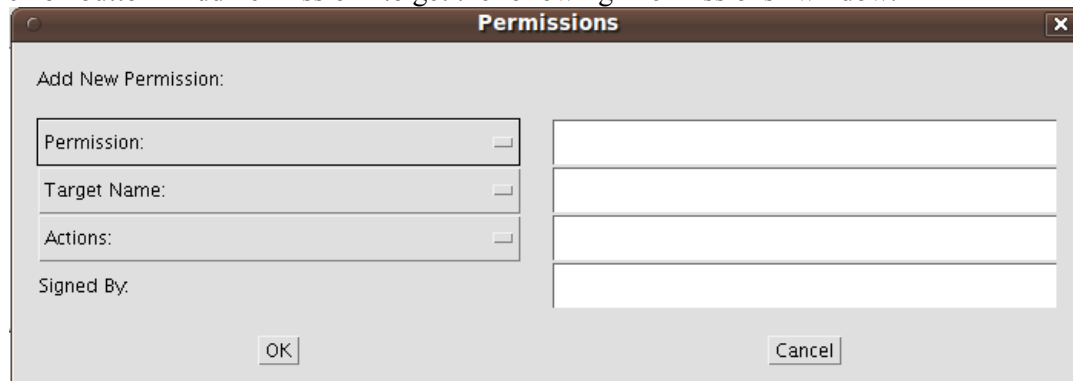
Policy Entry

CodeBase:

SignedBy:

Principals:

16. Click on button “Add Permission” to get the following “Permissions” window.



Permissions

Add New Permission:

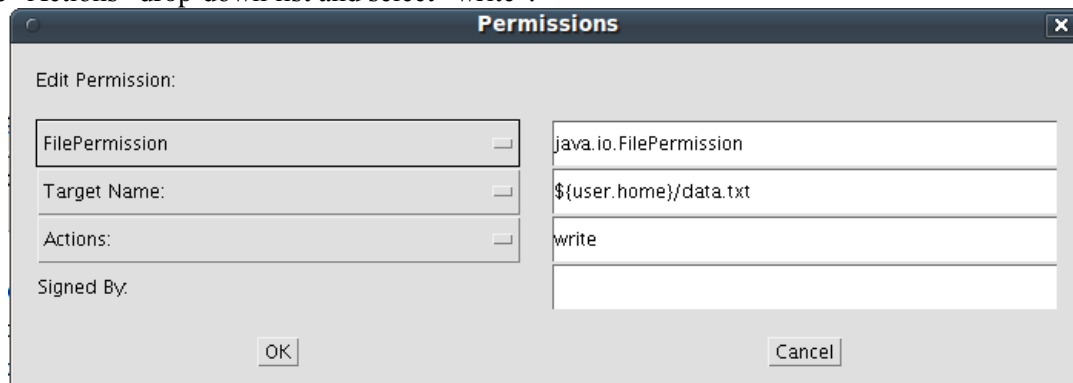
Permission:

Target Name:

Actions:

Signed By:

17. Click on the “Permission” drop-down list and select “FilePermission”. Type file name “\${user.home}/data.txt” in the textbox to the right of the “Target Name” drop-down list. Click on the “Actions” drop-down list and select “write”.



Permissions

Edit Permission:

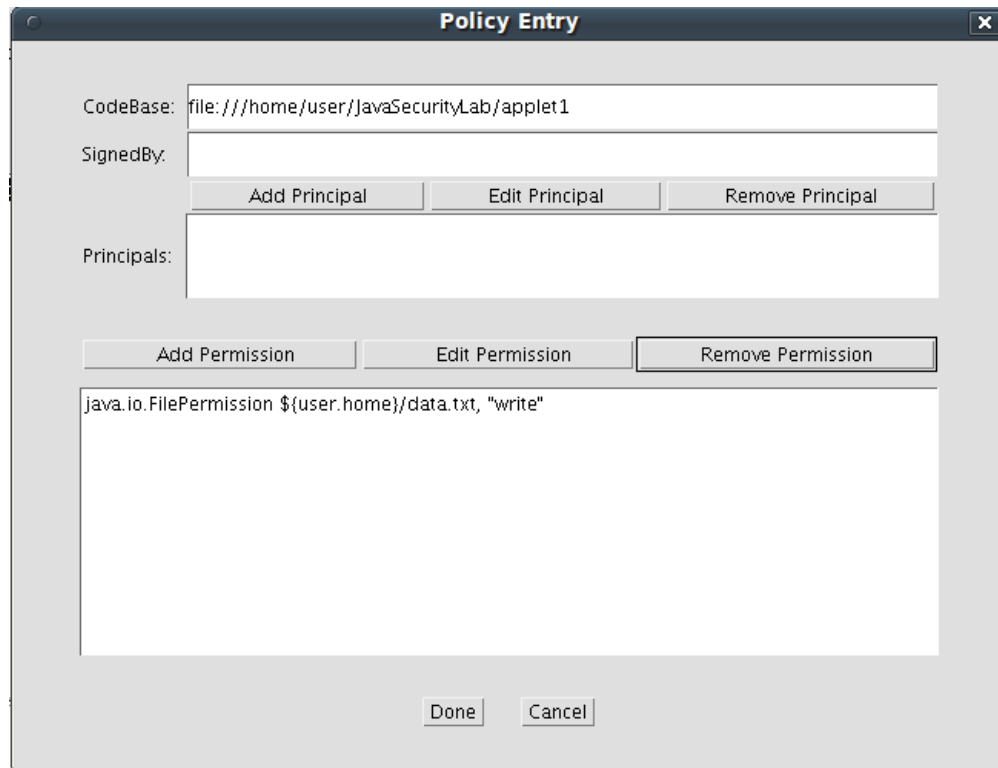
FilePermission

Target Name:

Actions:

Signed By:

18. Click on the “OK” button. The new permission displays in a line in the Policy Entry window.



Policy Entry

CodeBase: file:///home/user/JavaSecurityLab/applet1

SignedBy:

Add Principal Edit Principal Remove Principal

Principals:

Add Permission Edit Permission Remove Permission

java.io.FilePermission \${user.home}/data.txt, "write"

Done Cancel

19. Now you will add the second policy to allow the applet to read the value of Java property “user.home”. Click on the “Add Permission” button again to launch the Permissions window. Choose “PropertyPermission” for “Permission”, type “user.home” for “Target name”, choose “read” for “Actions”, and then click on the OK button to close the Permissions window.



Permissions

Add New Permission:

PropertyPermission java.util.PropertyPermission

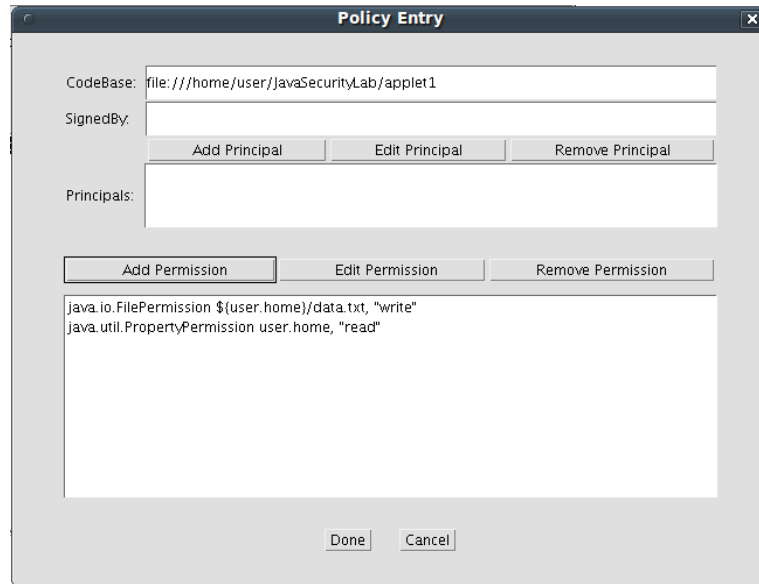
Target Name: user.home

read read

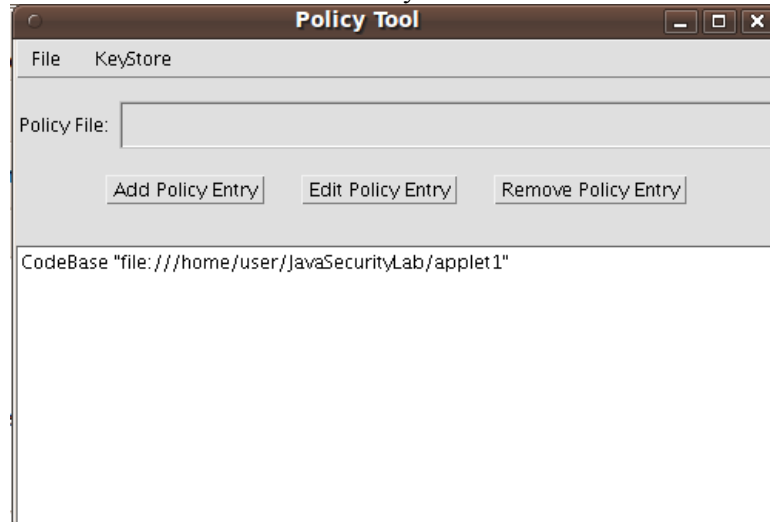
Signed By:

OK Cancel

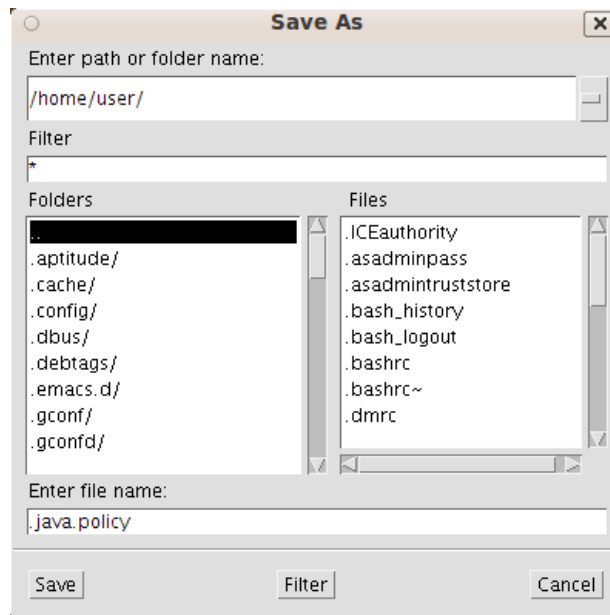
20. Now you see the following Policy Entry window:



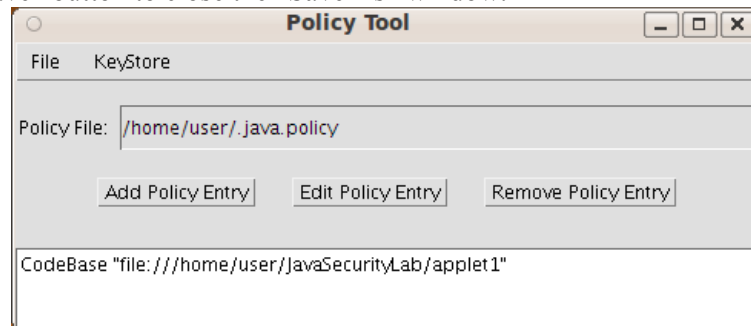
21. Click on the “Done” button to return to the “Policy Tool” window.



22. Click on menu item “File|Save As”, enter “/home/user/” followed by the Enter key in the “Enter path or folder name” textbox, and enter “.java.policy” in the “Enter file name” textbox.



23. Click on the “Save” button to close the “Save As” window.



24. Click on menu item “File|Exit” to shut down the Policy Tool window.

25. Run “more ~/.java.policy” to review the contents of the Java policy file:

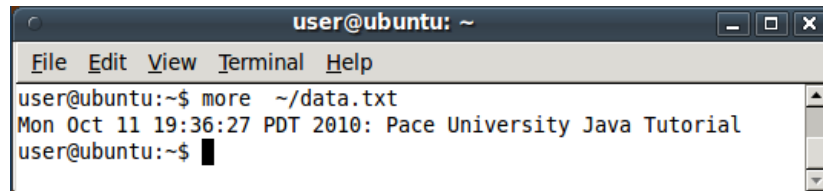
```
grant codeBase "file:///home/user/JavaSecurityLab/applet1" {
    permission java.io.FilePermission "${user.home}/data.txt", "write";
    permission java.util.PropertyPermission "user.home", "read";
};
```

If you know the syntax for granting rights to Java code, you could directly edit the policy file. The utility “policytool” is optional to make the task of granting rights easier.

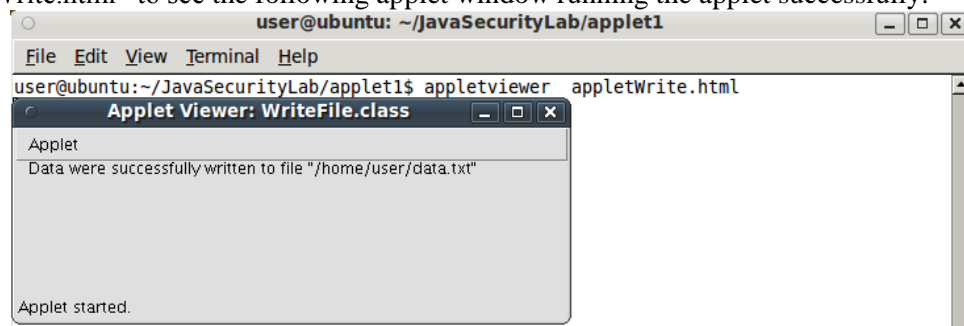
26. In folder “/home/user/JavaSecurityLab/applet1”, double click on file “appletWrite.html” and display it in a web browser, and this time you see the applet message “Data were successfully written to file “/home/user/data.txt” ”.



27. Run “more ~/data.txt” to review the contents of file “data.txt”.



28. Now you will launch the web page with Java utility “appletviewer” again. Run “appletviewer appletWrite.html” to see the following applet window running the applet successfully:



Now you have successfully granted security rights to an applet located at a specific location so it could write to a local file.

Question 23: Should you use relative path or absolute path to specify CodeBase?

Question 24: Should you use relative path or absolute path to specify a file as the value of the *Target Name* of the *Permissions* window?

Question 25: What is the meaning of “\${user.home}”?

4.6 Granting Special Rights to Applets Based on Code Signing

In the last lab you learned how to grant special rights to applets based on code location. In this lab you learn how to do the same thing based on which private key has signed the code. You will learn how to make a Jar file of Java classes, sign the Jar file with a private key, write an HTML file and use the Jar file to provide code for an applet, and grant file write to any Java code signed by the particular private key.

First let us recall what you have done in your second lab in this tutorial. You have created a pair of private/public keys in keystore file “~/JavaSecurityLab/PaceKeystore”, assigned alias “PaceKey” to this pair of keys, exported the self-signed digital certificate for the public key, imported the digital certificate for the public key into another keystore file “~/JavaSecurityLab/receiverKeystore”, and assigned alias “Pace” to this certificate in the latter keystore. In this lab the imaginary developer will use the private key with alias “PaceKey” in keystore file “~/JavaSecurityLab/PaceKeystore” to sign the Jar file of the applet class, and the imaginary user will grant special rights to all programs signed by the private key whose corresponding public key is in a digital certificate with alias “Pace” in keystore file “~/JavaSecurityLab/receiverKeystore”.

1. Launch the Ubuntu VM with username “user” and password 123456.
2. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.
3. Run “cd ~/JavaSecurityLab/applet2” to change work folder to “~/JavaSecurityLab/applet2”.
4. Run “gedit WriteFile.java” to review the contents of the applet source file. This file is the same as the one that you used in the last lab.
5. Run “javac WriteFile.java” to compile the source code into bytecode file “WriteFile.class”.
6. Run “jar cvf WriteFile.jar WriteFile.class” to create a Jar file “WriteFile.jar”. A Jar file is basically a zip file containing a set of Java classes and resources. The Jar format is easier for deployment of a set of related Java classes on a computer.
7. Run “7z l WriteFile.jar” to review the contents of file “WriteFile.jar” (the 7z command-line switch “l” means listing) .
8. Now assuming you were the developer, and you use your private key with alias “PaceKey” to sign the Jar file. Run the following command

```
jarsigner -keystore ../PaceKeystore -signedjar signedWriteFile.jar
WriteFile.jar PaceKey
```

on the same line to use the private key with alias “PaceKey” in keystore file “../PaceKeystore” to sign the jar file “WriteFile.jar” and write the output in file “signedWriteFile.jar”. When asked for keystore passphrase (password), enter “PaceUniversity”. When asked for key passphrase, enter “Seidenberg”. A new file “signedWriteFile.jar” will be generated.

9. Run “7z l signedWriteFile.jar” and note the new files in the Jar file for the digital signature of the original Jar file “WriteFile.jar”.
10. Run “gedit appletWrite2.html” to review its contents:

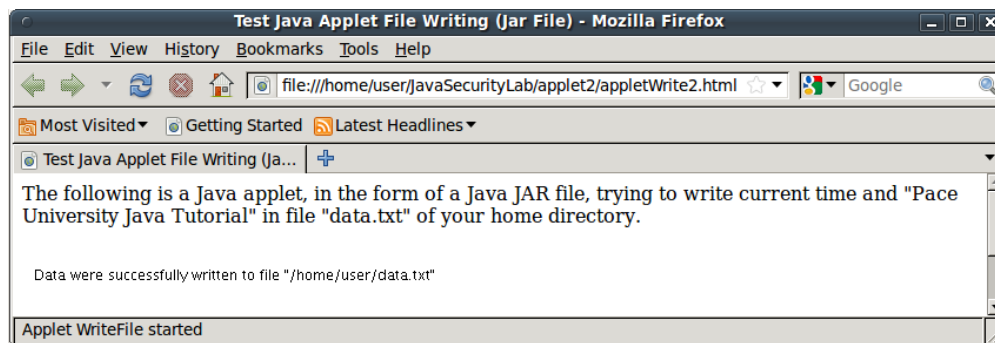
```
<html>
<head>
<title>Test Java Applet File Writing (Jar File)</title>
</head>
<body>
<p>The following is a Javaapplet, in the form of a Java JAR file, trying
to write the current time and "Pace University Java Tutorial" in file
"data.txt" of your home directory.</p>
<br/>
<applet code="WriteFile.class" width=420 height=100
archive="signedWriteFile.jar">
</applet>
</body>
</html>
```

If the applet class is contained in a Jar file, you can use the “archive” attribute of the “applet” tag to specify the Jar file.

11. In the file explorer, double-click on file “appletWrite2.html” to display it in the web browser. A security warning window will pop up:

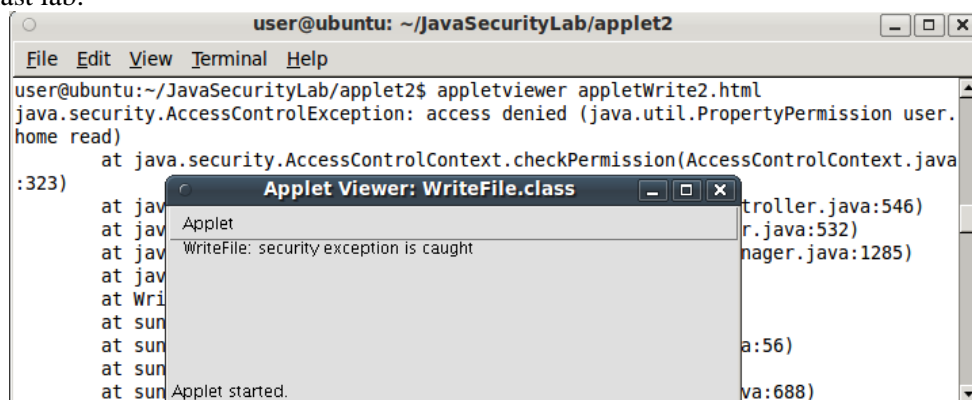


12. Click on the “Run” button to tell your computer that you trust this applet signed by John Smith, and the web browser will write data to file “~/data.txt” successfully.



If you shut down the web browser and repeat this step, you will be warned against this applet again. If you check the “Always trust content from this publisher” checkbox, your computer will remember that you always trust programs signed by the private key of John Smith and it will not warn you again before it runs programs signed by the same private key. But for this lab we will not check this checkbox. Declaring that you trust programs signed by a particular private key this way is too dangerous. You want to use the Java security policy to define what rights that you want to grant programs signed by this particular private key.

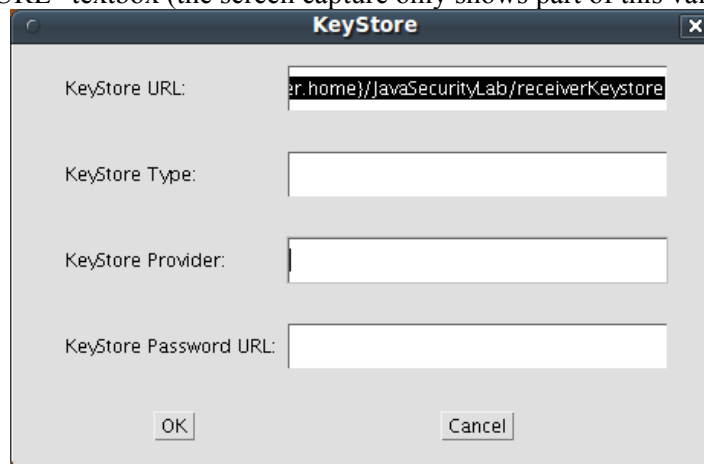
13. Shut down the web browser.
14. Run “appletviewer appletWrite2.html” and you get the security warnings similar to what you got in the last lab:



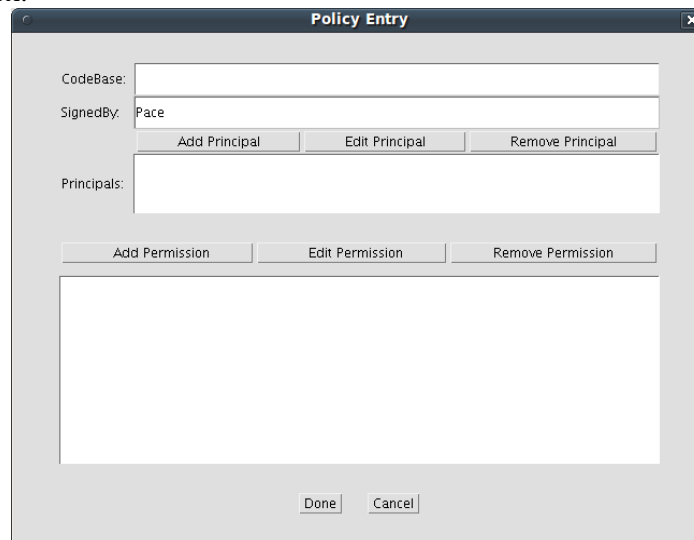
15. Now you are ready to grant access rights for all programs signed by this particular private key. Run “policytool” to display the following window.



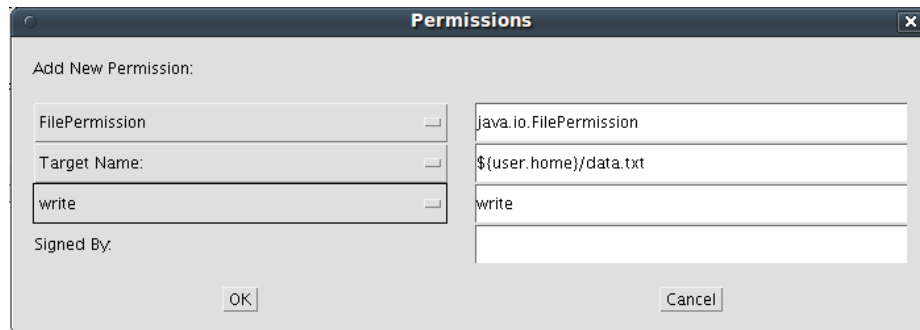
16. You first need to assign a keystore file for your policy file. Click on menu item “KeyStore|Edit” to pop up the “KeyStore” window, and enter “\${user.home}/JavaSecurityLab/receiverKeystore” in the “KeyStore URL” textbox (the screen capture only shows part of this value).



17. Click on the “OK” button to close the “KeyStore” window.
18. Click on the “Add Policy Entry” button to launch the “Policy Entry” window. Enter “Pace” in the “SignedBy” textbox.



19. Click on the “Add Permission” button to pop up the “Permissions” window. Choose “FilePermission” for “Permission”, enter “\${user.home}/data.txt” for “Target Name”, and choose “write” for “Actions”.



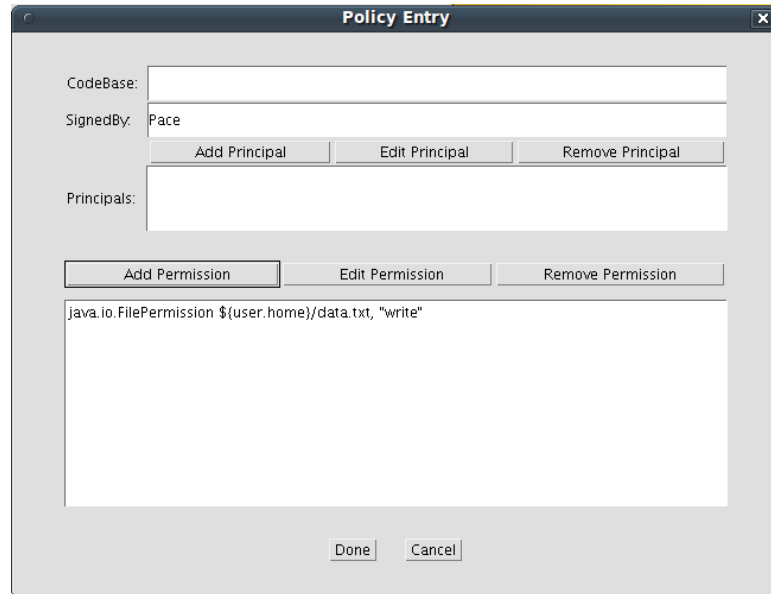
Permissions

Add New Permission:

FilePermission	java.io.FilePermission
Target Name:	\${user.home}/data.txt
write	write
Signed By:	

OK Cancel

20. Click on the “OK” button to close the window.



Policy Entry

CodeBase:

SignedBy: Pace

Add Principal Edit Principal Remove Principal

Principals:

Add Permission Edit Permission Remove Permission

java.io.FilePermission \${user.home}/data.txt, "write"

Done Cancel

29. Click on the “Add Permission” button again to pop up the “Permissions” window again. Choose “PropertyPermission” for “Permission”, type “user.home” for “Target name”, choose “read” for “Actions”, and then click on the OK button to close the Permissions window.



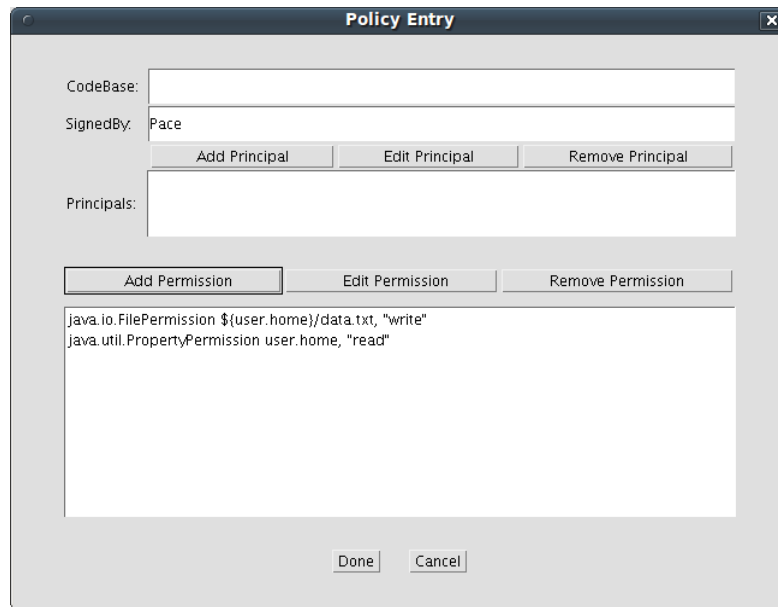
Permissions

Add New Permission:

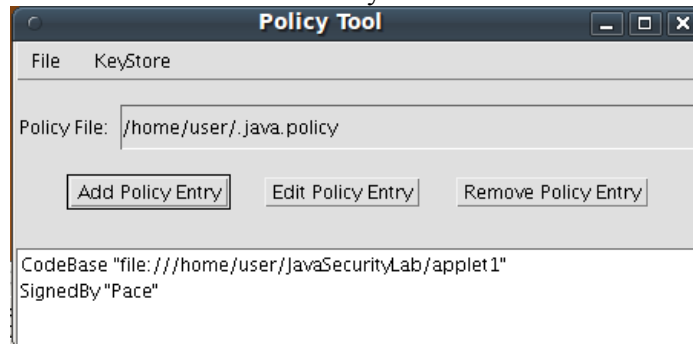
PropertyPermission	java.util.PropertyPermission
Target Name:	user.home
read	read
Signed By:	

OK Cancel

21. Now you see the following Policy Entry window:



22. Click on the “Done” button to return to the “Policy Tool” window.



23. Click on menu item “File|Save” to save the new policy, and click on menu item “File|Exit” to close the “Policy Tool” window.

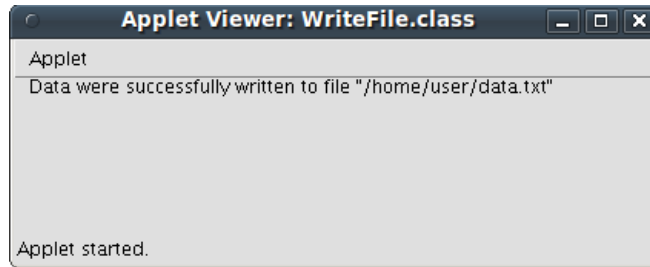
24. Run “more ~/.java.policy” to review the current contents of the policy file.

```
keystore "/home/user/JavaSecurityLab/receiverKeystore", "jks";

grant codeBase "file:///home/user/JavaSecurityLab/applet1" {
    permission java.io.FilePermission "${user.home}/data.txt", "write";
    permission java.util.PropertyPermission "user.home", "read";
};

grant signedBy "Pace" {
    permission java.io.FilePermission "${user.home}/data.txt", "write";
    permission java.util.PropertyPermission "user.home", "read";
};
```

25. Now run “appletviewer appletWrite2.html” again and the applet will successfully write data in file “~/data.txt”, as show by the screen capture below.



This concludes the lab for granting security rights based on code signing. You are encouraged to try to use both `CodeBase` and `SignedBy` to grant security rights to signed programs at specific file system locations.

Question 26: What is the implication if you check “Always trust content from this publisher” in a security warning window when you try to display a web page?

Question 27: Why “appletviewer” is a preferred tool for studying and developing web applications containing signed applets?

4.7 Creating a Certificate Chain to Implement a Trust Chain

In this lab you will work as a CA (Certificate Authority) to sign another certificate `SchoolKey`. You will show that if your computer trusts code signed by the CA, it will also trust code signed by `SchoolKey`. Therefore the certificate chain supports a trust chain. You will use a Java program “`SignCertificate.java`” to sign a certificate with the private key associated with another certificate. Due to the complexity we will not explain this program and just use it as a tool.

1. Launch the Ubuntu VM with username “user” and password 123456.
2. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.
3. Run “`cd ~/JavaSecurityLab/cert-chain`” to change the work folder to “~/JavaSecurityLab/cert-chain”.
4. Run “`javac SignCertificate.java`” to compile the program into bytecode file “`SignCertificate.class`”. You will see 39 warnings for the compilation, but it is harmless.
5. Run “`more WriteFile.java`” and “`more appletWrite2.html`” to review the contents of the two files. They are the same ones that you used in the last lab.
6. Run “`javac WriteFile.java`” to compile the program into bytecode file “`WriteFile.class`”. This is the same class you used in the previous lab.
7. Run “`jar cvf WriteFile.jar WriteFile.class`” to create the Jar file, as you did in the last lab.
8. Now you are going to function as the CA (certificate authority) to create the CA’s public/private key pair and its corresponding self-certified certificate in a new keystore with name “keystore” in the current folder. Run the following command on the same line:

```
keytool -genkey -alias CA -keypass 123456 -keystore keystore -storepass 123456 -
validity 365 -keyalg RSA
```

Argument “-validity 365” specifies that the certificate would be valid for 365 days. Argument “-keyalg RSA” specifies that the key generation algorithm would be RSA. Here you use trivial password 123456 to ease your typing. You would type “Certificate Authority” for the first and last names, “ITS” for organizational unit, “Pace University” for organization, “New York” for city, “NY” for state, and “US” for country code.

```
user@ubuntu: ~/JavaSecurityLab/cert-chain
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab/cert-chain$ keytool -genkey -alias CA -keypass 123456 -keystore keystore -storepass 123456 -validity 365 -keyalg RSA
What is your first and last name?
[Unknown]: Certificate Authority
What is the name of your organizational unit?
[Unknown]: ITS
What is the name of your organization?
[Unknown]: Pace University
What is the name of your City or Locality?
[Unknown]: New York
What is the name of your State or Province?
[Unknown]: NY
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY, C=US correct?
[no]: y
user@ubuntu:~/JavaSecurityLab/cert-chain$
```

9. Now you will function as an IT staff of Seidenberg School of Pace University to create the master public/private key pair and certificate for the School in the same keystore. You will associate the keys and certificate with alias “SchoolKey”. Run the following command:

```
keytool -genkey -alias SchoolKey -keypass 123456 -keystore keystore -storepass 123456 -validity 365 -keyalg RSA
```

You would type “Lab” for the first and last names, “Seidenberg School” for organizational unit, “Pace University” for organization, “New York” for city, “NY” for state, and “US” for country code.

```
user@ubuntu: ~/JavaSecurityLab/cert-chain
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab/cert-chain$ keytool -genkey -alias SchoolKey -keypass 123456 -keystore keystore -storepass 123456 -validity 365 -keyalg RSA
What is your first and last name?
[Unknown]: Lab
What is the name of your organizational unit?
[Unknown]: Seidenberg School
What is the name of your organization?
[Unknown]: Pace University
What is the name of your City or Locality?
[Unknown]: New York
What is the name of your State or Province?
[Unknown]: NY
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=Lab, OU=Seidenberg School, O=Pace University, L=New York, ST=NY, C=US correct?
[no]: y
user@ubuntu:~/JavaSecurityLab/cert-chain$
```

10. Run “java SignCertificate keystore CA SchoolKey SchoolKeySigned” to use the private key with alias “CA” to sign the certificate with alias “SchoolKey”, and associate alias “SchoolKeySigned” with the signed certificate in the keystore. Enter 123456 for all the three required passwords.

```

user@ubuntu: ~/JavaSecurityLab/cert-chain
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab/cert-chain$ java SignCertificate keystore CA Sc
hoolKey SchoolKeySigned
Keystore password: 123456
CA (CA) password: 123456
Cert (SchoolKey) password: 123456
user@ubuntu:~/JavaSecurityLab/cert-chain$

```

11. Run “keytool -export -alias SchoolKeySigned -keystore keystore -file SchoolKeySigned.cert” to export the certificate associated with “SchoolKeySigned” into file “SchoolKeySigned.cert”. Upon request enter 123456 for keystore password.
12. Run “keytool -import -alias SchoolKey -keystore keystore -file SchoolKeySigned.cert” to import the certificate in file “SchoolKeySigned.cert” back in the keystore entry associated with alias “SchoolKey”. Upon request enter 123456 for keystore password.
13. Run “keytool -list -v -keystore keystore” to review the contents of the keystore. Upon request enter 123456 as keystore password. You will see the following output:

```

user@ubuntu:~/JavaSecurityLab/cert-chain$ keytool -list -v -keystore keystore
Enter keystore password: 123456

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 3 entries

Alias name: schoolkeysigned
Creation date: Oct 18, 2010
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Lab, OU=Seidenberg School, O=Pace University, L=New York, ST=NY, C=US
Issuer: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY,
C=US
Serial number: 4cbd3110
Valid from: Mon Oct 18 22:48:00 PDT 2010 until: Tue Oct 18 22:48:00 PDT 2011
Certificate fingerprints:
  MD5:  CD:FD:F7:48:6E:A1:95:DD:87:76:02:5F:08:DA:3B:8B
  SHA1:  F5:6E:E4:FA:E1:C6:62:9A:0F:E8:43:C1:17:43:98:A2:18:E2:B3:B8
  Signature algorithm name: MD5withRSA
  Version: 3

*****
*****

Alias name: ca
Creation date: Oct 18, 2010
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY, C=US
Issuer: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY,
C=US
Serial number: 4cbd2e03
Valid from: Mon Oct 18 22:34:59 PDT 2010 until: Tue Oct 18 22:34:59 PDT 2011
Certificate fingerprints:
  MD5:  0B:B7:B1:48:36:30:FE:DB:E3:EA:35:17:6C:02:9C:C0
  SHA1:  BF:E3:0C:C1:1B:4C:36:BD:41:81:24:C4:7A:E2:2A:24:F6:4A:89:B9
  Signature algorithm name: SHA1withRSA
  Version: 3

```

```

*****
*****

Alias name: schoolkey
Creation date: Oct 18, 2010
Entry type: PrivateKeyEntry
Certificate chain length: 2
Certificate[1]:
Owner: CN=Lab, OU=Seidenberg School, O=Pace University, L=New York, ST=NY, C=US
Issuer: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY,
C=US
Serial number: 4cbd3110
Valid from: Mon Oct 18 22:48:00 PDT 2010 until: Tue Oct 18 22:48:00 PDT 2011
Certificate fingerprints:
  MD5:  CD:FD:F7:48:6E:A1:95:DD:87:76:02:5F:08:DA:3B:8B
  SHA1:  F5:6E:E4:FA:E1:C6:62:9A:0F:E8:43:C1:17:43:98:A2:18:E2:B3:B8
  Signature algorithm name: MD5withRSA
  Version: 3
Certificate[2]:
Owner: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY, C=US
Issuer: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY,
C=US
Serial number: 4cbd2e03
Valid from: Mon Oct 18 22:34:59 PDT 2010 until: Tue Oct 18 22:34:59 PDT 2011
Certificate fingerprints:
  MD5:  0B:B7:B1:48:36:30:FE:DB:E3:EA:35:17:6C:02:9C:C0
  SHA1:  BF:E3:0C:C1:1B:4C:36:BD:41:81:24:C4:7A:E2:2A:24:F6:4A:89:B9
  Signature algorithm name: SHA1withRSA
  Version: 3

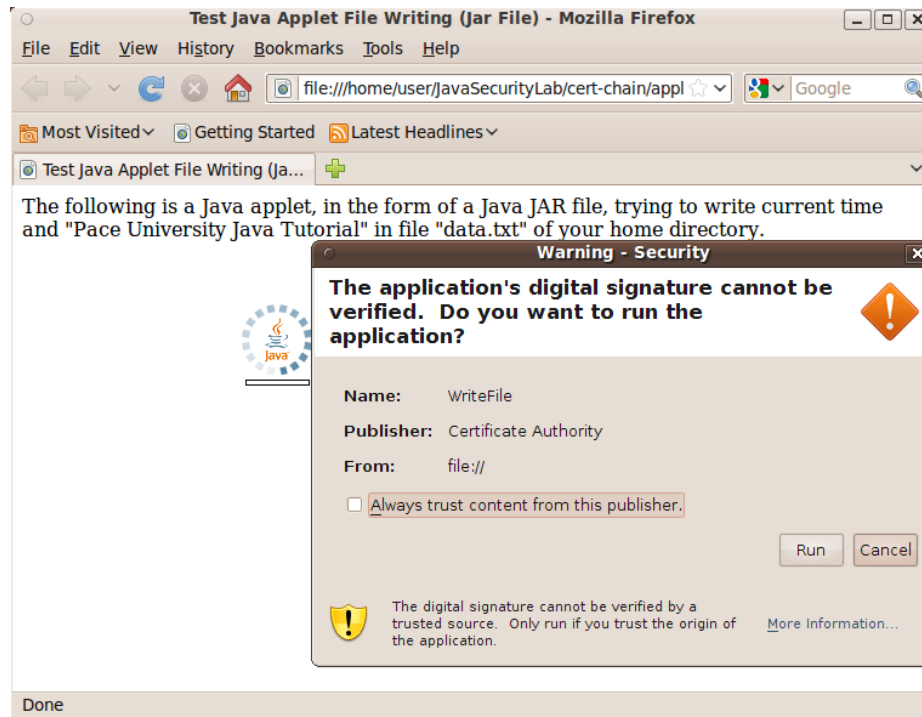
*****
*****

user@ubuntu:~/JavaSecurityLab/cert-chain$

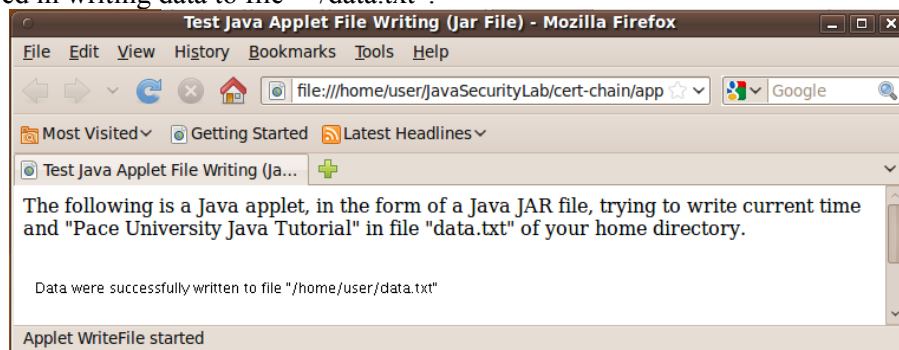
```

You can see that there are two certificates associated with alias “SchoolKey”. The first one is for the CA’s private key to sign the “SchoolKey” certificate. The second one is for the CA private key to sign its own certificate. This is your chain of certificates.

14. Now run “jarsigner -keystore keystore -signedjar signedWriteFile.jar WriteFile.jar CA” to sign the Jar file with the private key of the CA. The signed Jar file is “signedWriteFile.jar”. Upon request enter 123456 for keystore password.
15. Double-click on file “appleteWrite2.html” in file explorer and choose to display it. You will be asked whether you trust the applet signed by CA.



16. Check “Always trust content from this publisher” and click on the “Run” button, and the applet will succeed in writing data to file “~/data.txt”.



17. Run “`jarsigner -keystore keystore -signedjar signedWriteFile.jar WriteFile.jar SchoolKey`” to resign the Jar file with the private key associated with “SchoolKey”. Upon request enter 123456 for keystore password.
18. Double-click on file “appleteWrite2.html” in file explorer and choose to display it. You will NOT be asked whether you trust the applet signed by SchoolKey, and the applet will succeed in writing data to file “~/data.txt”. This is because you already told the computer that you always trust code signed by the CA. Since the certificate for “SchoolKey” is signed by the CA, your trust to the CA is transfer to your trust to the certificate for “SchoolKey”.

This lab provides clear idea and hands-on knowledge on how a CA signs your certificates, and how a certificate chain implements a trust chain. While your chain has only two certificates, it could be extended to any length.

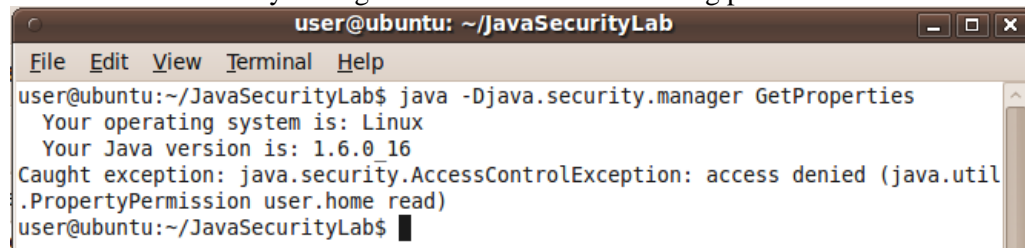
4.8 Protecting Your Computer from Insecure Java Applications

Sometimes you need to run an application but you don't have full trust on it. You need to find out what are the minimum resources that this application needs to access and in which form to access (read or write or both). Then you can run the application under control of the Java security manager to limit its access to most resources on your system, and explicitly grant it access to those necessary resources to obtain the desired service from the application with reduced danger to your computer. In this lab you will use the Java program "GetProperties.java" as the insecure Java application.

19. Launch the Ubuntu VM with username "user" and password 123456.
20. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
21. Run "cd ~/JavaSecurityLab" to change the work folder to "~/JavaSecurityLab".
22. Run "gedit GetProperties.java" to review the source code of file "GetProperties.java".
23. Run "javac GetProperties.java" to compile the source code into bytecode file "GetProperties.class".
24. Run "java GetProperties" to execute file "GetProperties.class" without the control of a Java security manager. You will see the following printout:

```
Your operating system is: Linux
Your Java version is: 1.6.0_16
Your user home directory is: /home/user
Your JRE installation directory is: /home/user/tools/jdk1.6.0_16/jre
Your Java extension directories are:
/home/user/tools/jdk1.6.0_16/jre/lib/ext:/usr/java/packages/lib/ext
```

25. Run "java -Djava.security.manager GetProperties" to execute file "GetProperties.class" under the control of a Java security manager. You will see the following printout:



Now the class runs as an applet and it cannot access Java property "user.home".

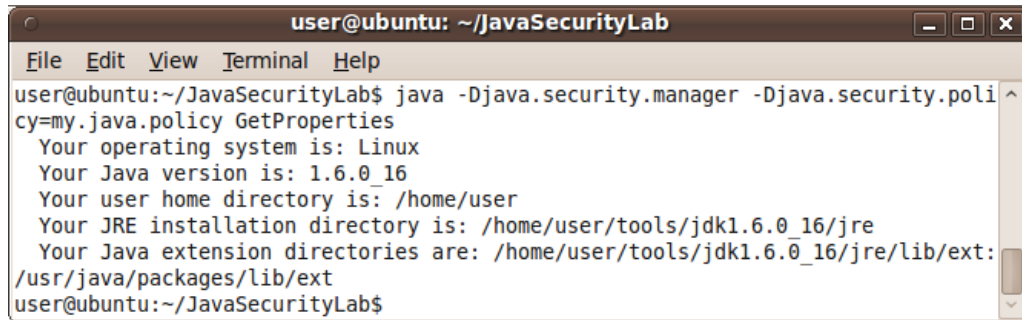
26. Run "more my.java.policy" to review the contents of file "my.java.policy":

```
grant codeBase "file:///home/user/JavaSecurityLab/" {
    permission java.util.PropertyPermission "user.home", "read";
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "java.ext.dirs", "read";
};
```

While you normally specify special access rights of applications in the default user's Java policy file "~/.java.policy", as you have done so far, you could also save part of the policies in a separate file and later apply the policies in this file by specifying this separate file as a Java command line argument, as demonstrated in the next step.

27. Run the following command to run the Java application "GetProperties" under the control of Java security manager, and grant the access rights to the application as specified in the Java policy file "my.java.policy":

```
java -Djava.security.manager -Djava.security.policy=my.java.policy GetProperties
```



```
user@ubuntu: ~/JavaSecurityLab
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab$ java -Djava.security.manager -Djava.security.policy=my.java.policy GetProperties
Your operating system is: Linux
Your Java version is: 1.6.0_16
Your user home directory is: /home/user
Your JRE installation directory is: /home/user/tools/jdk1.6.0_16/jre
Your Java extension directories are: /home/user/tools/jdk1.6.0_16/jre/lib/ext:/usr/java/packages/lib/ext
user@ubuntu:~/JavaSecurityLab$
```

As you can see, this time the application runs successfully. You could also copy the contents of file “my.java.policy” into “~/.java.policy” so you don’t need to specify a Java policy file on Java command line.

This concludes the lab for explaining how to run a Java application under the control of a Java security manager so it runs in (almost) isolation, and use a Java policy file to selectively grant rights to the application for it to deliver its services.

Question 28: Why a user should bother to run a Java application using the Java security manager?

Question 29: How to run a Java application under the supervision of the Java security manager?

Question 30: What is the recommended way to study or develop multiple Java programs using different and maybe conflicting security policies?

4.9 Securing File Exchange with Java Security API and Newly Created Keys


Java has a security API that allows a Java program to carry out some of the tasks normally conducted by Java security utilities. These tasks include generating public/private key pairs, exporting and importing these keys to/from files or keystores, and generating digital certificates and signatures. The coming three labs guide you to explore these functions against the background of secure file exchange against identity attacks or contents modification. If you also need to maintain the confidentiality of the file contents, you can use one of the Java ciphers of the Java Cryptography Extension/Architecture (JCE, <http://download.oracle.com/javase/1.4.2/docs/guide/security/CryptoSpec.html>, <http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html>) to encode the file before its signing and decode the file after it has been verified for the correct signature. These three labs differ in where the private key comes from. In the current lab, the private key is dynamically created. In the following labs, the private key is retrieved from a file or from a keystore.

The objective of these three labs is to give you the first impression and main idea of the Java security API and how does it work. You should not try to understand each line of the code unless you have taken two Java courses.

In this lab, you need to securely send a data file, “GettysburgAddress.txt”, to another person. You will first use a Java program to create a pair of public/private keys, generate the digital signature of the data file, and export the public key and the signature into files. The receiver of the data file will use the second

Java program to read in the public key, the signature and the data file to verify whether the signature is correct.

1. Launch the Ubuntu VM with username “user” and password 123456.
2. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.
3. Run “cd ~/JavaSecurityLab/JavaSecurityAPI/use-new-key” to change work folder to “~/JavaSecurityLab/JavaSecurityAPI/use-new-key”.
4. Run “gedit GenerateSignature.java &” to review the contents of Java source code file “GenerateSignature.java”.
5. Run “javac GenerateSignature.java” to compile the file into bytecode file “GenerateSignature.class”.
6. Run “gedit VerifySignature.java &” to review its contents of Java source code file “VerifySignature.java”.
7. Run “javac VerifySignature.java” to compile the file into bytecode file “VerifySignature.class”.
8. Run “java GenerateSignature GettysburgAddress.txt” to sign data file “GettysburgAddress.txt”, and generate the public key file “public-key” and signature file “signature”.
9. Run “java VerifySignature public-key signature GettysburgAddress.txt” to verify the signature for data file “GettysburgAddress.txt”. You will see that the signature verification is successful.



```
user@ubuntu: ~/JavaSecurityLab/JavaSecurityAPI/use-new-key
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/use-new-key$ javac GenerateSignature.java
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/use-new-key$ javac VerifySignature.java
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/use-new-key$ java GenerateSignature GettysburgAddress.txt
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/use-new-key$ java VerifySignature public-key signature GettysburgAddress.txt
signature verifies: true
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/use-new-key$
```

Now we list important segments of the programs and explain them.

4.9.1 Java command-line arguments

```
class GenerateSignature {
    public static void main(String[] args) ... { ..... }
}
```

When a Java program runs, it first runs the main() method. The main method has an array of strings, args[...], as its parameter for holding command-line arguments. When you run “java Prog a b c”, args[0] will hold string “a”, args[1] will hold string “b”, and args[2] will hold string “c”.

4.9.2 Generating public/private keys

```
1. KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
2. SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
3. keyGen.initialize(1024, random);
4. KeyPair pair = keyGen.generateKeyPair();
5. PrivateKey priv = pair.getPrivate();
6. PublicKey pub = pair.getPublic();
7. byte[] key = pub.getEncoded();
8. FileOutputStream keyfos = new FileOutputStream("public-key");
9. keyfos.write(key);
10. keyfos.close();
```

1. This line creates a key generator object, sets its key generator algorithm to be DSA (digital signature algorithm) implemented by SUN (now Oracle).
2. This line creates a secure random number generator object, and sets its random number algorithm to be SHA1PRNG implemented by SUN. The key generator needs random numbers to generate random keys.
3. This line specifies that the key generator will use the random generator created on line 2 and generate keys of length 1024 bytes.
4. This line generates the key pair.
5. This line retrieves the private key.
6. This line retrieves the public key.
7. Lines 7-10 are for writing the public key in file “public-key”. Line 7 saves the key in an array of bytes encoded for input/output.
8. This line opens file “public-key” for output.
9. This line writes the key bytes out into the file.
10. This line closes the file to reserve system resources.

4.9.3 Generating a digital signature

```

1. Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
2. dsa.initSign(priv);
3. FileInputStream fis = new FileInputStream(args[0]);
4. BufferedInputStream bufin = new BufferedInputStream(fis);
5. byte[] buffer = new byte[1024];
6. while (bufin.available() != 0) {
7.     int len = bufin.read(buffer);
8.     dsa.update(buffer, 0, len);
9. };
10. bufin.close();
11. byte[] realSig = dsa.sign();
12. FileOutputStream sigfos = new FileOutputStream("signature");
13. sigfos.write(realSig);
14. sigfos.close();

```

1. This line creates a signature generator object, sets its signing algorithm to be SHA1 (a type of hash function for generating fingerprints) with DSA (digital signature algorithm) implemented by SUN (now Oracle).
2. This line specifies the private key for the signature generator.
3. This line opens the file whose name is specified by the first command-line argument (the first string after the class name that you run).
4. This line provides the opened file with a data buffer so you could know whether you still have data to read on line 6 before you actually read the data on line 7.
5. This line creates an array for 1024 bytes as a data buffer. Since you use a private key of 1024 bytes long, you will process 1024 bytes a time to generate the fingerprint.
6. Lines 6-9 are a loop. Code “bufin.available()” returns number of bytes in the buffer that have not been read yet. Code on lines 7 and 8 will be executed repeatedly until the buffer is empty.
7. This line reads up to 1024 bytes into the 1024-byte buffer (the last read may have less than 1024 bytes). The actual number of read bytes is saved in variable “len”.
8. This line applies the DSA algorithm on the newly read data.
9. This line marks the end of the loop.
10. This line closes the file to release the file buffer.
11. This line actually generates the signature of the input data and save it in an array of bytes.
12. Lines 12-14 are for writing the signature out in a file named “signature”.

4.10 Securing File Exchange with Java Security API and Keys in Files

Typically you generate a pair of public/private key pairs and use it for multiple times. For example you may generate a pair of keys to sign all of your documents, and another pair to sign your programs. You can either save the key pairs in files or a keystore. In this lab you will save the key pair in files for reuse.

1. Launch the Ubuntu VM with username “user” and password 123456.
2. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.
3. Run “cd ~/JavaSecurityLab/JavaSecurityAPI/key-in-file” to change work folder to “~/JavaSecurityLab/JavaSecurityAPI/key-in-file”.
4. Run “gedit GenerateKeys.java &” to review the contents of Java source code file “GenerateKeys.java”. This program generates a pair of public/private keys, and save them in files “public-key” and “private-key” for reuse.
5. Run “javac GenerateKeys.java” to compile the file into bytecode file “GenerateKeys.class”.
6. Run “gedit GenerateSignature2.java &” to review the contents of Java source code file “GenerateSignature2.java”. This program import the private key from file “private-key”, use it to sign the data file, and write the resulting signature in file “signature”.
7. Run “javac GenerateSignature2.java” to compile the file into bytecode file “GenerateSignature2.class”.
8. Run “gedit VerifySignature.java &” to review the contents of Java source code file “VerifySignature.java”. This is the same file that you used in the last two labs.
9. Run “javac VerifySignature.java” to compile the file into bytecode file “VerifySignature.class”.
10. Run “java GenerateKeys” to generate the key pair in files “public-key” and “private-key”.
10. Run “java GenerateSignature2 private-key GettysburgAddress.txt” to sign data file “GettysburgAddress.txt” with the private key in file “private-key”, and save the generated signature in file “signature”.
11. Run “java VerifySignature public-key signature GettysburgAddress.txt” to verify the signature for data file “GettysburgAddress.txt”. You will see that the signature verification is successful.



```
user@ubuntu: ~/JavaSecurityLab/JavaSecurityAPI/key-in-file
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-file$ javac GenerateKeys.java
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-file$ javac GenerateSignature2.java
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-file$ javac VerifySignature.java
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-file$ java GenerateKeys
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-file$ java GenerateSignature2 private-key Ge
ttysburgAddress.txt
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-file$ java VerifySignature public-key signa
ture GettysburgAddress.txt
signature verifies: true
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-file$
```

4.10.1 Importing a private key from a file

```
1. FileInputStream keyfis = new FileInputStream(args[0]);
2. byte[] encKey = new byte[keyfis.available()];
3. keyfis.read(encKey);
4. keyfis.close();
5. PKCS8EncodedKeySpec privKeySpec = new PKCS8EncodedKeySpec(encKey);
6. KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
7. PrivateKey priv = keyFactory.generatePrivate(privKeySpec);
```

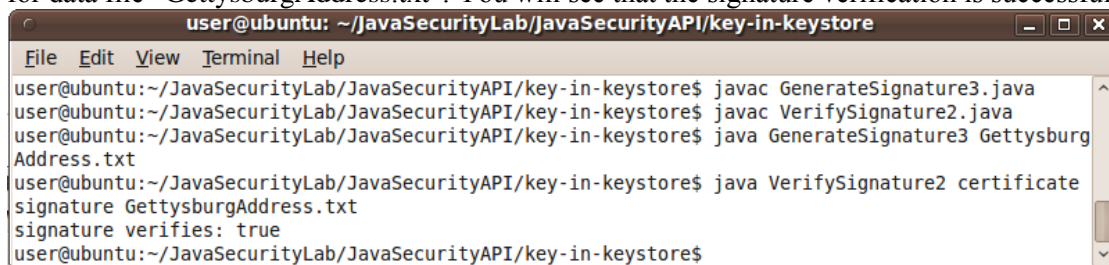
1. Line 1 opens the private key file whose name is specified by the first command-line argument to “java” (the first string after the class name).

2. Line 2 checks the data size in the file, and creates an array of bytes of the same size.
3. Line 3 read the contents of the file into the array of bytes.
4. Line 4 closes the file for recycling the resources associated with the file.
5. Line 5 generates a PKCS8 encoding of the private key. When you write keys to files, the public keys are encoded in the X509 format, and the private keys are encoded in the PKCS8 format.
6. Line 6 creates a key factory object using the DSA algorithm implemented by SUN.
7. Line 7 creates the private key based on the PKCS8 specification of the key.

4.11 Securing File Exchange with Java Security API and Keys in a Keystore

In this lab you will repeat what you did in the last two labs but use the digital certificate and private key, associated with alias “PaceKey”, in PaceKeystore. You created this keystore and the key/certificate in an earlier lab.

1. Launch the Ubuntu VM with username “user” and password 12345678.
2. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.
3. Run “cd ~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore” to change work folder to “~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore”.
4. Run “gedit GenerateSignature3.java &” to review the contents of Java source code file “GenerateSignature3.java”.
5. Run “javac GenerateSignature3.java” to compile the file into bytecode file “GenerateSignature3.class”.
6. Run “gedit VerifySignature2.java &” to review the contents of Java source code file “VerifySignature2.java”.
7. Run “javac VerifySignature2.java” to compile the file into bytecode file “VerifySignature2.class”.
8. Run “java GenerateSignature3 GettysburgAddress.txt” to sign data file “GettysburgAddress.txt” with the private key with alias “PaceKey” in keystore “PaceKeystore”, write the signature in file “signature”, and write the certificate in file “certificate”.
9. Run “java VerifySignature2 certificate signature GettysburgAddress.txt” to verify the signature for data file “GettysburgAddress.txt”. You will see that the signature verification is successful.



```

user@ubuntu: ~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore
File Edit View Terminal Help
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore$ javac GenerateSignature3.java
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore$ javac VerifySignature2.java
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore$ java GenerateSignature3 Gettysburg
Address.txt
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore$ java VerifySignature2 certificate
signature GettysburgAddress.txt
signature verifies: true
user@ubuntu:~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore$

```

4.11.1 Importing a private key and certificate pair from a keystore

```

1. KeyStore ks = KeyStore.getInstance("JKS");
2. FileInputStream ksfis = new FileInputStream("../PaceKeystore");
3. BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
4. ks.load(ksbufin, "PaceUniversity".toCharArray());
5. PrivateKey priv = (PrivateKey) ks.getKey("PaceKey",
    "Seidenberg".toCharArray());
6. java.security.cert.Certificate cert = ks.getCertificate("PaceKey");
7. byte[] encodedCert = cert.getEncoded();
8. FileOutputStream certfos = new FileOutputStream("certificate");

```

```
9. certfos.write(encodedCert);
10. certfos.close();
```

1. Line 1 creates a keystore object of type JKS (Java Keystore). The keystores for Java have nothing to do with the keystores for PGP.
2. Line 2 opens the keystore file “PaceKeystore” two levels up in the file system.
3. Line 3 provides the file with a data buffer.
4. Line 4 loads the contents of the keystore file into the in-memory keystore object with the keystore password “PaceUniversity”. Since the load() method needs an array of characters for the password, we first converted the string “PaceUniversity” into an array of characters.
5. Line 5 retrieves the private key associated with alias “PaceKey”; and it provides the keystore object with the password “Seidenberg” for the key.
6. Line 6 retrieves the certificate in the keystore object associated with alias “PaceKey”. You may note that you don’t need to provide a key password to retrieve a certificate because a certificate is public.
7. Lines 7-10 write the certificate out into file “certificate”. Line 7 gets a copy of the certificate encoded for input/output.
8. Line 8 opens file “certificate” for output.
9. Line 9 writes the encoded certificate in file “certificate”.
10. Line 10 closes the file for recycling the resources associated with the file.

4.11.2 Importing a public key from a digital certificate file

```
1. FileInputStream certfis = new FileInputStream(args[0]);
2. java.security.cert.CertificateFactory cf =
   java.security.cert.CertificateFactory.getInstance("X.509");
3. java.security.cert.Certificate cert = cf.generateCertificate(certfis);
4. PublicKey pubKey = cert.getPublicKey();
```

1. Line 1 opens the digital certificate’s file specified as the first command-line argument to command “java” (the first string after the class name).
2. Line 2 creates a certificate factory object of type “X.509”.
3. Line 3 uses the certificate factory object to generate a certificate object based on the contents of the certificate file.
4. Line 4 retrieves the public key from the certificate object.

5 Review Questions

Question 31: Why Java security is important?

Question 32: What are the most vulnerable folders for Java security?

Question 33: Why applets always run using Java security manager?

Question 34: List three resources that programs using Java security manager cannot access by default?

Question 35: Why you should bother to run some applications using Java security manager?

Question 36: How to selectively grant access rights to applets or applications?

6 Appendix

6.1 File “GetProperties.java”

```
class GetProperties {
    public static void main(String[] args) {
        String s;
        try {
            s = System.getProperty("os.name", "not specified");
            System.out.println(" Your operating system is: " + s);
            s = System.getProperty("java.version", "not specified");
            System.out.println(" Your Java version is: " + s);
            s = System.getProperty("user.home", "not specified");
            System.out.println(" Your user home directory is: " + s);
            s = System.getProperty("java.home", "not specified");
            System.out.println(" Your JRE installation directory is: " + s);
            s = System.getProperty("java.ext.dirs", "not specified");
            System.out.println(" Your Java extension directories are: " + s);
        } catch (Exception e) {
            System.err.println("Caught exception: " + e);
        }
    }
}
```

6.2 File “WriteFile.java”

```
import java.awt.*;
import java.applet.*;
import java.io.*;

public class WriteFile extends Applet {
    public void paint(Graphics g) {
        try {
            String fileName = System.getProperty("user.home") +
                System.getProperty("file.separator") + // / or \
                "data.txt";
            File f = new File(fileName);
            PrintWriter output = new PrintWriter(new FileWriter(f), true); // auto-flush
            output.println(new java.util.Date() + ": Pace University Java Tutorial");
            g.drawString("Data were successfully written to file \"" + fileName + "\"",
                10, 10);
        }
        catch (SecurityException e) {
            g.drawString("WriteFile: security exception is caught", 10, 10);
            e.printStackTrace(); // print error messages to terminal window for debugging
        }
        catch (IOException ioe) {
            g.drawString("WriteFile: I/O exception is caught", 10, 10);
        }
    }
}
```

6.3 File “appletWrite.html”

```
<html>
<head>
<title>Test Java Applet File Writing</title>
</head>
<body>
<p>The following is a Java applet trying to write the current time
and "Pace University Java Tutorial" in file "data.txt" of your home
folder.</p>
<br/>
<applet code="WriteFile.class" width=420 height=100>
</applet>
</body>
</html>
```

6.4 File “appletWrite2.html”

```
<html>
<head>
<title>Test Java Applet File Writing (Jar File)</title>
</head>
<body>
<p>The following is a Java applet, in the form of a Java JAR file, trying to write
current time and "Pace University Java Tutorial" in file "data.txt" of your home
directory.</p>
<br/>
<applet code="WriteFile.class" width=420 height=100 archive="signedWriteFile.jar">
</applet>
</body>
</html>
```

6.5 File “my.java.policy”

```
grant codeBase "file:///home/user/JavaSecurityLab/" {
    permission java.util.PropertyPermission "user.home", "read";
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "java.ext.dirs", "read";
};
```

6.6 File “GenerateSignature.java”

```
import java.io.*;
import java.security.*;

// Generate a pair of public/private keys, generate signature of the input data
// with the private key, write out the signature and the public key in files
// "signature" and "public-key".
class GenerateSignature {
    public static void main(String[] args) throws Exception {
```

```

    if (args.length != 1) {
        System.out.println("Usage: java GenerateSignature fileToSign");
        System.exit(-1);
    }
    // Generate a key pair
    // set key generator algorithm DSA implemented by SUN
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
    // set random number algorithm SHA1PRNG implemented by SUN
    SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
    keyGen.initialize(1024, random); // set key length be 1024 bytes
    KeyPair pair = keyGen.generateKeyPair(); // generate key pair
    PrivateKey priv = pair.getPrivate();
    PublicKey pub = pair.getPublic();
    // Create a Signature object and initialize it with algorithm SHA1withDSA
    // implemented by SUN.
    Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
    dsa.initSign(priv); // set the private key
    // Read in data, up to 1024 bytes at a time, to generate fingerprint
    // args[0]: first command-line argument, data file name
    FileInputStream fis = new FileInputStream(args[0]); // open data file
    // provide buffer for data input
    BufferedInputStream bufIn = new BufferedInputStream(fis);
    byte[] buffer = new byte[1024];
    while (bufIn.available() != 0) { // while there are still unread data
        // read up to 1024 bytes, set actual length in len
        int len = bufIn.read(buffer);
        dsa.update(buffer, 0, len); // apply hash to the recent data segment
    };
    bufIn.close();
    // Generate a signature by encoding the fingerprint with the private key
    byte[] realSig = dsa.sign();
    // Save the signature in file "signature"
    FileOutputStream sigfos = new FileOutputStream("signature");
    sigfos.write(realSig);
    sigfos.close();
    // Save the public key in file "public-key"
    byte[] key = pub.getEncoded();
    FileOutputStream keyfos = new FileOutputStream("public-key");
    keyfos.write(key);
    keyfos.close();
};
}

```

6.7 File “VerifySignature.java”

```

import java.io.*;
import java.security.*;
import java.security.spec.*;

// Import public key and signature from the files, and use them to authenticate
// the author and validate the contents of the data file
class VerifySignature {
    public static void main(String[] args) throws Exception {
        if (args.length != 3) {
            System.out.println("Usage: java VerifySignature publicKeyFile" +
                " signatureFile dataFile");
            System.exit(-1);
        }
        // import encoded public key
        FileInputStream keyfis = new FileInputStream(args[0]);

```

```

// create a byte array as large as the file contents
byte[] encKey = new byte[keyfis.available()];
keyfis.read(encKey);
keyfis.close();
// use file contents to create a public key specification object
X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);
// create a key factory using algorithm DSA implemented by SUN
KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
// recover public key from its specification
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
// input the signature bytes
FileInputStream sigfis = new FileInputStream(args[1]);
byte[] sigToVerify = new byte[sigfis.available()];
sigfis.read(sigToVerify);
sigfis.close();
// create a Signature object using algorithm SHA1withDSA implemented by SUN
Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
// initialize the Signature object with the public key
sig.initVerify(pubKey);
// Read in data, up to 1024 bytes a time, to generate fingerprint
// args[2]: third command-line argument, data file name
FileInputStream datafis = new FileInputStream(args[2]); // open data file
// provide buffer for data input
BufferedInputStream bufin = new BufferedInputStream(datafis);
byte[] buffer = new byte[1024];
int len;
while (bufin.available() != 0) { // while there are still unread data
    // read up to 1024 bytes, set actual length in len
    len = bufin.read(buffer);
    sig.update(buffer, 0, len); // apply hash to the recent data segment
};
bufin.close();
// decode the input signature into fingerprint with the public key,
// compare it with the fingerprint just created above.
boolean verifies = sig.verify(sigToVerify);
System.out.println("signature verifies: " + verifies);
}
}

```

6.8 File “GenerateKeys.java”

```

import java.io.*;
import java.security.*;

// Generate a pair of public/private keys, and save them in files "public-key" and
// "private-key".
class GenerateKeys {
    public static void main(String[] args) throws Exception {
        // Generate a key pair
        // set key generator algorithm DSA implemented by SUN
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
        // set random number algorithm SHA1PRNG implemented by SUN
        SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
        keyGen.initialize(1024, random); // set key length be 1024 bytes
        KeyPair pair = keyGen.generateKeyPair(); // generate key pair
        PrivateKey priv = pair.getPrivate();
        PublicKey pub = pair.getPublic();
        // Save the public key in file "public-key"
        byte[] key = pub.getEncoded();
    }
}

```

```

        FileOutputStream keyfos = new FileOutputStream("public-key");
        keyfos.write(key);
        keyfos.close();
        // Save the private key in file "private-key"
        key = priv.getEncoded();
        keyfos = new FileOutputStream("private-key");
        keyfos.write(key);
        keyfos.close();
    };
}

```

File “GenerateSignature2.java”

```

import java.io.*;
import java.security.*;
import java.security.spec.*;

// Import a private key from a file, generate signature of the input data with the
// private key, and write out the signature in file "signature".
class GenerateSignature2 {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.println("Usage: java GenerateSignature2 privateKeyFile" +
                               " fileToSign");
            System.exit(-1);
        }
        // Import the encoded private key from file args[0]
        FileInputStream keyfis = new FileInputStream(args[0]);
        // create a byte array as large as the file contents
        byte[] encKey = new byte[keyfis.available()];
        keyfis.read(encKey);
        keyfis.close();
        // use file contents to create a private key specification object
        PKCS8EncodedKeySpec privKeySpec = new PKCS8EncodedKeySpec(encKey);
        // create a key factory using algorithm DSA implemented by SUN
        KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
        // recover private key from its specification
        PrivateKey priv = keyFactory.generatePrivate(privKeySpec);
        // Create a Signature object and initialize it with algorithm SHA1withDSA
        // implemented by SUN.
        Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
        dsa.initSign(priv); // set the private key
        // Read in data, up to 1024 bytes a time, to generate fingerprint
        // args[1]: second command-line argument, data file name
        FileInputStream fis = new FileInputStream(args[1]); // open data file
        // provide buffer for data input
        BufferedInputStream bufin = new BufferedInputStream(fis);
        byte[] buffer = new byte[1024];
        int len;
        while (bufin.available() != 0) { // while there are still unread data
            // read up to 1024 bytes, set actual length in len
            len = bufin.read(buffer);
            dsa.update(buffer, 0, len); // apply hash to the recent data segment
        };
        bufin.close();
        // Generate a signature by encoding the fingerprint with the private key
        byte[] realSig = dsa.sign();
        // Save the signature in a file "signature"
        FileOutputStream sigfos = new FileOutputStream("signature");
        sigfos.write(realSig);
        sigfos.close();
    };
}

```

```
}
```

6.9 File “GenerateSignature2.java”

```
import java.io.*;
import java.security.*;
import java.security.spec.*;

// Import a private key from a file, generate signature of the input data with the
// private key, and write out the signature in file "signature".
class GenerateSignature2 {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.println("Usage: java GenerateSignature2 privateKeyFile " +
                               " fileToSign");
            System.exit(-1);
        }
        // Import the encoded private key from file args[0]
        FileInputStream keyfis = new FileInputStream(args[0]);
        // create a byte array as large as the file contents
        byte[] encKey = new byte[keyfis.available()];
        keyfis.read(encKey);
        keyfis.close();
        // use file contents to create a private key specification object
        PKCS8EncodedKeySpec privKeySpec = new PKCS8EncodedKeySpec(encKey);
        // create a key factory using algorithm DSA implemented by SUN
        KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
        // recover private key from its specification
        PrivateKey priv = keyFactory.generatePrivate(privKeySpec);
        // Create a Signature object and initialize it with algorithm SHA1withDSA
        // implemented by SUN.
        Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
        dsa.initSign(priv); // set the private key
        // Read in data, up to 1024 bytes a time, to generate fingerprint
        // args[1]: second command-line argument, data file name
        FileInputStream fis = new FileInputStream(args[1]); // open data file
        // provide buffer for data input
        BufferedInputStream bufin = new BufferedInputStream(fis);
        byte[] buffer = new byte[1024];
        int len;
        while (bufin.available() != 0) { // while there are still unread data
            // read up to 1024 bytes, set actual length in len
            len = bufin.read(buffer);
            dsa.update(buffer, 0, len); // apply hash to the recent data segment
        };
        bufin.close();
        // Generate a signature by encoding the fingerprint with the private key
        byte[] realSig = dsa.sign();
        // Save the signature in file "signature"
        FileOutputStream sigfos = new FileOutputStream("signature");
        sigfos.write(realSig);
        sigfos.close();
    };
}
```

6.10 File “GenerateSignature3.java”

```
import java.io.*;
```

```

import java.security.*;
import java.security.spec.*;

// Import a private key from a keystore, generate signature of the input data with
// the private key, and write out the signature in file "signature".
class GenerateSignature3 {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.out.println("Usage: java GenerateSignature3 fileToSign");
            System.exit(-1);
        }
        // Import the private key and certificate with alias "PaceKey" from keystore
        // "../PaceKeystore"
        KeyStore ks = KeyStore.getInstance("JKS"); // get an empty keystore object
        // open keystore file
        FileInputStream ksfis = new FileInputStream("../PaceKeystore");
        // load keystore contents in buffer
        BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
        // load keystore contents into the keystore object; supply keystore password
        ks.load(ksbufin, "PaceUniversity".toCharArray());
        // retrieve private key for alias "PaceKey" with key password "Seidenberg"
        PrivateKey priv = (PrivateKey)ks.getKey("PaceKey", "Seidenberg".toCharArray());
        // Retrieve the corresponding certificate and save it in a file
        java.security.cert.Certificate cert = ks.getCertificate("PaceKey");
        byte[] encodedCert = cert.getEncoded(); // encode the certificate for I/O
        // save the certificate in a file named "certificate" */
        FileOutputStream certfos = new FileOutputStream("certificate");
        certfos.write(encodedCert);
        certfos.close();
        // Create a Signature object and initialize it with algorithm SHA1withDSA
        // implemented by SUN.
        Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
        dsa.initSign(priv); // set the private key
        // Read in data, up to 1024 bytes a time, to generate fingerprint
        // args[0]: first command-line argument, data file name
        FileInputStream fis = new FileInputStream(args[0]); // open data file
        // provide buffer for data input
        BufferedInputStream bufin = new BufferedInputStream(fis);
        byte[] buffer = new byte[1024];
        int len;
        while (bufin.available() != 0) { // while there are still unread data
            // read up to 1024 bytes, set actual length in len
            len = bufin.read(buffer);
            dsa.update(buffer, 0, len); // apply hash to the recent data segment
        }
        bufin.close();
        // Generate a signature by encoding the fingerprint with the private key
        byte[] realSig = dsa.sign();
        // Save the signature in file "signature"
        FileOutputStream sigfos = new FileOutputStream("signature");
        sigfos.write(realSig);
        sigfos.close();
    }
}

```

File “VerifySignature2.java”

```

import java.io.*;
import java.security.*;
import java.security.spec.*;

// Import certificate and signature from the files, get public key from

```

```

// certificate, and use them to authenticate the author and validate the contents
// of the data file
class VerifySignature2 {
    public static void main(String[] args) throws Exception {
        if (args.length != 3) {
            System.out.println("Usage: java VerifySignature2 certificateFile" +
                               " signatureFile dataFile");
            System.exit(-1);
        }
        // Import a digital certificate from args[0]
        FileInputStream certfis = new FileInputStream(args[0]);
        java.security.cert.CertificateFactory cf =
            java.security.cert.CertificateFactory.getInstance("X.509");
        java.security.cert.Certificate cert = cf.generateCertificate(certfis);
        // retrieve public key from the certificate
        PublicKey pubKey = cert.getPublicKey();
        // input the signature bytes
        FileInputStream sigfis = new FileInputStream(args[1]);
        byte[] sigToVerify = new byte[sigfis.available()];
        sigfis.read(sigToVerify);
        sigfis.close();
        // create a Signature object using algorithm SHA1withDSA implemented by SUN
        Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
        // initialize the Signature object with the public key
        sig.initVerify(pubKey);
        // Read in data, up to 1024 bytes a time, to generate fingerprint
        // args[2]: third command-line argument, data file name
        FileInputStream datafis = new FileInputStream(args[2]); // open data file
        // provide buffer for data input
        BufferedInputStream bufin = new BufferedInputStream(datafis);
        byte[] buffer = new byte[1024];
        int len;
        while (bufin.available() != 0) { // while there are still unread data
            // read up to 1024 bytes, set actual length in len
            len = bufin.read(buffer);
            sig.update(buffer, 0, len); // apply hash to the recent data segment
        };
        bufin.close();
        // decode the input signature into fingerprint with the public key,
        // compare it with the fingerprint just created above.
        boolean verifies = sig.verify(sigToVerify);
        System.out.println("signature verifies: " + verifies);
    }
}

```