Ser321 Principles of Distributed Software Systems

7. Security in Java2 Platform
7.a Security Background

7.a.1 Motivation and Objectives

• How can you electronically receive information and be confident who it's from and that the information didn’t change in transit? (Digital Signature)

• How can you electronically send information between applications so that it can’t be read and understood by anyone else in transit? (Encryption)

• How can you control what system resources executing Java code can access?

• Objectives

  - To be able to control access to system resources for Java2 local and remote applications. To understand define and utilize security Policy files.

  - To know how to generate, examine and utilize digital signatures. To manage public and private keys with both the key tool and the API.

  - To use cryptography services to encrypt and decrypt information.

  - To understand how to use secure sockets in Java for communication between distributed components of an application.
7.a.2 References for Java Security

- **On-Line References**
  - In the Java Tutorial, see the Specialized Trail on Security
    https://docs.oracle.com/javase/tutorial/
  - See the Sockets and Encryption sections of Essentials of Java Part 2
    http://www.oracle.com/technetwork/java/basicjava2-138746.html
7.a.3 Overview

- **Digital Signatures, Keys and Certificates**
  - Information transferred over the internet is received in the form it was sent and with assurance of who it came from and that it wasn’t changed enroute.
  - Services available in Java2 through J2sdk API and through tools.

- **Encryption and Decryption**
  - Information transferred is converted from clear text to unreadable cipher data before transfer.
  - Upon receipt, the receiver has key information that allows conversion back to clear text.
  - Multi-supplier JCA API and use of selected user API classes.

- **Controlling access to system resources** in an executing Java program - properties and policy files.
  - Resources include local files, connections, and properties.
  - Applications loaded through classpath.
  - Applets downloaded (via a jar file) from the internet or class files dynamically loaded.
7.b Significance of the Problem

7.b.1 Application and System Vulnerability

- Attacks ranging from server systems to small and home systems:
  - **Equifax** data breach of SSN’s birth dates & addresses cost up to $4 Billion.
  - **WannaCry** spread to over 100 countries in less than 24 hours. While the ransomware attackers only received $140,000, losses estimated at $4 Billion.
  - **Washington State University** had a hard drive stolen that contained the personal data of over 1 million people.
  - **Denial of Service attacks**: Usually do not compromise valuable information but can be source of considerable downgrade of service: These can be characterized as: “I wouldn’t drive my own car to rob a bank”

- **Trojan Horse** virus attacks
  - **Code Red** worm from 2001 cost $2.6 billion to clean up

- Defense against these attacks is **reactive (anti-virus) not proactive**
7.b.2 Denial of Service

- Any intrusion that interrupts the service of one or more users
  - If I program my cell phone to repeatedly call yours, I deny you service

- Distributed Denial of Service
  - If I program several other people’s cell phones to call yours

- Network Appliances
  - Home appliances, thermostat, lights, door locks, Parking lot gates
  - Printers
  - May have simple hard-wired access such as user-ids and passwords.
  - Source of Denial of Service attack by flooding the internet with traffic; such as: “I need to be rebooted”
7.b.3 Security Issues

- **E-Commerce** is deeper than Network Credit Card Transactions
  - Rapid growth in use of social media, messaging, email and other distributed groupware to exchange information that is financially or otherwise sensitive
  - Business agents are largely unaware of how insecure their communications really are
  - Added burden for the end-user

- **Application Development**
  - **Technologies** for providing security in an application have become much less complex to use
  - Slow to realize the potential for harm from distributed applications
  - Developers largely un-trained in use of security mechanisms

- **Need for a Culture Change**
  - **Certificates** for all applications running on your machine
  - Build checks into receipt of sensitive information (code or data)
  - Most organizations use a sandbox and don’t run executable attachments.
7.b.4 Distributed Objects and Security

- RMI built on much of Java’s rich execution environment
  - **Reflection**
  - **Serialization**
  - **Dynamic class loading** and subsequent execution
  - Added flexibility goes hand-in-hand with **added security risk**

- Current design **relies on the Developer** of Distributed Object Applications
  - Must explicitly build-in appropriate use of Java’s Security mechanisms.
7.c Signing Information

7.c.1 Message Digests

- A Hash function produces a **Message Digest**
  - Hash function, takes an arbitrary length input message and generates a fixed length digest

- **Properties of a hash function**
  - **One-way** hash from data source to a fixed-size digest -- you can not recover the data source from the message digest alone.
  - **Low probability** that different data sources will produce the same digest -- **collisions** are highly unlikely dependent upon the hash function used to generate the digest.
  - **Non-Revealing**: Digest does not reveal anything about the data source.
7.c.2 Message Digest Generation Algorithms

- Message Digest is the basis for generating and verifying Digital Signatures
- Hash Functions (algorithms)
  - MD2 Message Digest Version 2 (Rivest 1989) a 128 bit (16 bytes) digest
  - MD4 Message Digest Version 4 (Rivest 1990) a 128 bit (16 bytes) digest
  - MD5 Message Digest Version 5 (1991) a 128 bit (16 bytes) digest
  - MD5 Described: https://www.ietf.org/rfc/rfc1321.txt
- SHS (Secure Hash Standard) coordinated by NIST Information Technology Laboratory:
  - SHA-1 generates 160-bit digest (80 bit security against collision attacks)
  - SHA-256 generates 256-bit digest (128 bit security against collision)
  - SHA-512 generates a 512-bit digest (256 bits of security, and a 384-bit hash may be obtained by truncating the SHA-512 output.)
7.c.3 Secure Hash Standard

- **Explanation**

  “This Standard specifies secure hash algorithms - SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256 - for computing a condensed representation of electronic data (message). When a message of any length less than 2**64 bits (for SHA-1, SHA-224 and SHA-256) or less than 2**128 bits (for SHA-384, SHA-512, SHA-512/224 and SHA-512/256) is input to a hash algorithm, the result is an output called a message digest. The message digests range in length from 160 to 512 bits, depending on the algorithm. Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits).”
7.c.4 Message Digests and the Java API

- Digital Signature and Digest examples can be found in the project digitalSign.jar
- JDK provides the SHA-1 algorithm
  - produces a 20 byte (digest) signature of data
  - probability of the same digest from distinct data sources is 1 in $2^{160}$
- The Java API also provides the MD2 and MD5 message digest algorithms as used in generating digital signatures.
- Message Digests form the basis for authentication and integrity
- Example DigestTest.java
  - javac -d classes src/DigestTest.java
  - java -cp classes ser321.security.DigestTest
7.c.5  Digital Signatures

• Used for **authentication** and **integrity** assurance of digital data.
  - “Verify who it came from and that its not changed in transit”
  - **Integrity** means the data has not been modified or tampered with
  - **Authenticity** means the data indeed comes from whoever claims to have created and signed it.

• Sender transfers both Message and Signature.

• Receiver Verifies message matches Signature using Public Key
7.c.6 Key Management

- Two types of key management
  - **Asymmetric** (Public/Private) keys (digital signatures and encryption)
  - **Symmetric** (secret) key algorithms (used for encrypted transmissions)

- **Asymmetric Keys** are created in matching pairs
  - **Private key** - used to generate signature (or decrypt cipher) and kept confidential to whoever is doing the signing.
  
  - **Public key** - used to verify authenticity of the message (or encrypt clear text to create a cipher). Distribute the public key to anyone who will receive signed information, or encrypt messages to be read by private key.

  - The mechanism for generating a key set and the relationship between public and private key is dependent on the signature algorithm used. The public key cannot spoof signature generation of the private key.

- **Managing keys**
  - Client application (browser) manages (public) keys.

  - Jdk provides a **keytool**, which manages **keystores** and **certificates**.

  - API to programmatically manage keys. (**java.security.KeyStore**)
7.c.7 Digital Signatures

- Receiver wants assurance of the message sender’s identity (**authentication**) and that the message has not been altered in transit (**integrity**).

- **Signing**
  - Generate a digest, noted as \( H(m) \), of the message to be sent: \( m \Rightarrow H(m) \)
  - Generate a signature (encryption of the digest) using the Private Key and an encryption algorithm: \( s = Encryption_{PrK}(H(m)) \)
  - Assuming the receiver already has the public key, send the message \( m \) and the signature \( s \) to the receiver

- **Verifying**
  - The receiver already has the matching public key (certificate); receives the message \( m \) and digital signature \( s \).
  - Decrypt the signature using the public key and matching algorithm to get a digest: \( H(m') = Decryption_{PbK}(s) \)
  - Generate a digest of the message received: \( m \Rightarrow H(m) \)
  - If \( H(m) = H(m') \) then the message is verified unchanged and from the private key owner; otherwise unverified.
7.c.8 Digital Signature Algorithms

- **Encryption/Decryption Algorithms** (Note, roles of public and private keys may be reversed in the literature as compared to the Java Tutorial)
  - **RSA** (Rivest-Shamir-Adleman)
  - **DSS** (Digital Signature Standard). A signature-only with underlying algorithm called **DSA** (Digital Signature Algorithm).

- There are primarily three algorithms (among many) for generating and verifying digital signatures:
  - **Digital Signature Algorithm** (DSA)
  - **RSA** (as specified in ANSI X9.31), and
  - **Elliptic Curve DSA** (ECDSA; as specified in ANSI X9.62)
  - All three techniques can make use of the Secure Hash Algorithm (SHA-1), which is specified in the Secure Hash Standard (SHS)
7.c.9 Using Java API - Signature Class

- A Signature object (see java.security) can be used to generate and verify digital signatures.
  - The application developer requests and uses an instance of the Signature engine class to get the functionality of a digital signature algorithm.

- Sun provides DSA and RSA in J2SDK
  - see java.security.Signature javadoc for algorithms provided by SUN
  - The DSA algorithm using the SHA-1 message digest algorithm can be specified as SHA1withDSA.
  - In the case of RSA, there are multiple choices for the message digest algorithm, so the signing algorithm could be specified as, for example, MD2withRSA, MD5withRSA, or SHA1withRSA.
7.c.10 Certificates

- Verifying digitally signed information (by the receiver) depends exclusively on the **receiver’s trust in the authenticity of the public key**.
  - Is this actually the public key of Fujitsu Software Corporation?

- **Certificates**
  - Provide a **chain of trust** that the public key belongs to whom you want to believe it belongs to.
  - A known entity certifies that a know entity certifies that a know entity certifies that ... this public key belongs to TimLindquist
  - A certificate is a digitally signed statement from one entity, saying that the public key of some other entity has a particular value.

- The best assurance is to be handed a physical copy of the public key (on a floppy or CD, for instance) by the signer.
  - Next best is to accept a certificate from a trusted certificating agency.
  - Certificating agencies require identification equivalent to notarization.

- **Public Certification Authorities**, include: [VeriSign](http://www.verisign.com), [Thawte](http://www.thawte.com), and [Entrust](http://www.entrust.com)
7.c.11 Keytool and Key Stores

- **keytool** manages a file of private keys and certificates. (Accessible via API)
- To create a new private/public key pair (defaults to DSA with 1024bit key)
  - `keytool -genkey -alias tim -keypass timpwd -keystore timStore`
  - `keytool -genkey -alias nick -keypass nickpwd -keystore nickStore`
- To export a public key certificate and to print information about a certificate
  - `keytool -export -alias nick -file nickcert.cer -keystore nickStore`
- To print information about a certificate file
  - `keytool -printcert -file nickcert.cer`
- To import a certificate
  - `keytool -import -alias nick -file nickcert.cer -keystore timStore`
- To print information about aliases in a keystore
  - `keytool -list [-alias tim] -keystore timStore`
- Other commands:
  - `keytool -genkey -alias tim -keyalg RSA -validity 120 -keystore tStore`
  - `-certreq` //generates a certificate signing request (CSR)
  - `-selfcert` //generates an X.509 self-signed certificate
7.c.12 Jar Signer Tool

• The jarsigner tool is used for two purposes:
  - to **digitally sign** Java ARchive (JAR) files, and
  - to **verify the digital signature** of signed JAR files.

• jarsigner uses key and certificate information from a keystore to generate
digital signatures for JAR files.

• **Signing a Jar file**
  - jarsigner -keystore nickStore -storepass nickpwd -keypass nickpwd gradeapplet.jar nick
  - When signing a jar file, you must specify the alias for the private key, the
    keystore (default is .keystore in user.home) and the jar file to be signed.

• Signing a jar produces two new files in the **META-INF** directory of the jar
  - **Signature file** (.SF) includes the name and digest for each file in the jar
  - **Signature block file** (.DSA) includes the signature of the .SF file and the
    Certificate (public key) of the private key used to sign the jar.

• **Verifying a signed Jar file**
  - jarsigner -verify -certs -verbose gradeapplet.jar
7.c.13 Using Java API - Certificates and Keystores

• See: `java.security.KeyStore` and `java.security.cert.Certificate`

• Loading a keystore from disk and getting a private key

```java
KeyStore ks = KeyStore.getInstance("JKS");
ks.load(new FileInputStream(storeFileName),keyPw.toCharArray());
PrivateKey pk = (PrivateKey)ks.getKey(user,keyPw.toCharArray());
```

• Importing a certificate from a file into a keystore

```java
FileInputStream fis = new FileInputStream(certFileName);
BufferedInputStream bis = new BufferedInputStream(fis);
CertificateFactory cf=CertificateFactory.getInstance("X.509");
java.security.cert.Certificate cert = cf.generateCertificate(bis);
ks.setCertificateEntry(alias, cert);
ks.store(new FileOutputStream(storeFileName),keyPw.toCharArray());
```
7.c.14 Using Java API - Signature Class

- Three phases to using a Signature object to sign or verify an information stream
  - Get a signature object specifying the hash function and digital signature algorithm, for example,
    - `Signature sig = Signature.getInstance("SHA1withRSA");`
    - `Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");`

- **Initialization**
  - Receiver: initializes for verification with a public key
  - Sender: initializes for signing with a private key and optional random number generator

- **Updating**
  - Successive calls to update the signature object as the data stream is processed (from file, for example)

- **Sign** or **verify** a signature based on all updated data stream bytes
  - `public boolean verify(byte[] signature) //signature match data?`
  - `public byte[] sign() //generate the signature byte stream`
7.c.15 Using Java API - Signature Class Example

- Assuming Key has been generated, such as:

  - keytool -genkey -alias tim -keyalg RSA -validity 120 -keystore tStore

- Generating a Digital Signature

  PrivateKey pk = (PrivateKey)ks.getKey(user,keyPw.toCharArray());
  Signature sig = Signature.getInstance("SHA1withRSA");
  sig.initSign(pk);
  sig.update(stringToSign.getBytes());
  byte[] ret = sig.sign();

- Verifying a Digital Signature

  PublicKey pk=ks.getCertificate(alias).getPublicKey();
  Signature sig = Signature.getInstance("SHA1withRSA");
  sig.initVerify(pk);
  String stringToVerify = . . .; byte[] digitalSignature = . . .;
  sig.update(stringToVerify.getBytes());
  if(sig.verify(digitalSignature)){ . . . }

- From the Digital Signature examples: digitalSign.jar

  - javac -d classes src/SignFile.java src/VerifyFile.java
  - java -cp classes ser321.security.SignFile myLet
  - java -cp classes ser321.security.VerifyFile pubKeyForLet sigOfmyLet myLet
7.c.16 Some Defaults for Jdk Security Tools

- Defaults for **KeyStore**
  - type is “**JKS**”
  - provider is “**SUN**”

- Defaults for Key
  - algorithm is **DSA**
  - key size is 1024

- Default certificate algorithm is **MD5**

- Default signature algorithms
  - if private key is of type "**DSA**", the default signature algorithm is "**SHA1withDSA**"
  - if private key is of type "**RSA**", the default signature algorithm is "**MD5withRSA**"
7.d  Controlling Java Applications and Applets

7.d.1  The Java2 Model for Accessing Resources

- From the Java tutorial: the model for controlling access to resources in Java2
7.d.2 Domains

- The **portions of a program** (classes) together with the **permissions granted** to those classes

- The classes making up a domain are specified by
  - where the code came from - **codebase http://java.sun.com**
  - who signed the code - **signedBy “Sun”**

- Permissions are granted to the code making up a domain
  - `grant signedBy “Sun”, codeBase “http://java.sun.com” {` 
    - `permission java.io.FilePermission “/tmp/-”, “read, write, delete”;
    - `permission java.util.PropertyPermission “*”, “read”;
    - `permission java.net.SocketPermission “*.sun.com”, “connect”;
  }

- `/tmp/-` means all files in `/tmp/` and recursively in all subdirectories of `tmp`
- `/tmp/*` would apply only to the file and subdirectory names in `tmp`
- May have one * in specification of a domain name, (*.sun.com), but it must be the leftmost character in the string

- See javadoc **java.security.Permission** and its subclasses
7.d.3 How are Domains Used?

• A running Java application may include classes loaded from many sources.
• Each class belongs to a protection domain based on origin and who signed it.
• When access to a system resource is attempted, every class on the call stack must be running in a protection domain that grants access to the resource. Otherwise a security exception is thrown.
• Problem: “Some” classes must be able to do their job independent of who calls them. (a class that plays a video clip, for example.)
• Solution:
  - A class may be Privileged (implements PrivilegedAction interface)
  - Any code within the run method of the PrivilegedAction will be able to use its permissions regardless of permissions of classes on the stack. `AccessController.doPrivileged(PrivilegedAction action)`
  - A privileged class may only assert a permission that it already has.
7.d.4 Example Policy Files to Control a Java Application

- **Problem Statement.** An executing java program has access to all of the files in and below the current directory (property user.dir) by default policy files. If an application, running the security manager, tries to access files above or outside of the current directory then a security exception will be thrown (unless a domain is established to explicitly allow the access).

- Example, **PrintKeyStore.java**, is an application that reads a keyStore and prints all aliases contained in the store. Two command line arguments
  - an absolute path to the keyStore file
  - the password for the keyStore
  - Consider: What happens when the path argument points outside the current directory?

- Consider the following contexts for executing **PrintKeyStore.java**
  - Without the security manager, -- **access is permitted**
  - With the security manager, but no permissions, -**throws security exception**
  - With the security manager and permission, -- **access permitted**
7.d.5 Example, without Security Manager

- To sign and provide an application the Signer must
  - All three scenarios are provided through ant in the example:
    - See: policyFiles.jar
- Download, extract and build the project. Edit the build.xml file for comment with directions for localizing the example found in readme.txt. You must:
  - create a keystore if the keystore/key in Data/timStore has expired.
  - edit the policy file changing the absolute paths for codebase and for the file permission. These must match your installation directory.
  - edit build.xml so the keystore, keyword and alias properties match your keystore.
  - Execute the three scenarios with execute, executeWithSecMgr and executeWithPolicy targets
- Execute PrintKeyStore without the security manager
  - java -cp classes ser321.security.PrintKeyStore Data/timStore timword
7.d.6 Example, with Security Manager, but no Policy

- Executing with security manager, but without a protection domain defined to grant access to files generates an exception
  
  - `java -cp . -Djava.security.manager -jar lib/PrintKeyStore Data/timStore timword`

- Note that this execution should throw an exception:
  
  - `java.security.AccessControlException`
  - access denied `java.io.FilePermission`
  - `read` access to file: `...\Data\timStore`
  - on the statement: `new FileInputStream(args[0])`
7.d.7 Example, with Security Manager and Policy File

• The Ant build file creates a jar file containing the classes and places it in the
  lib directory:

• The policy file (policy) must reflect your install directory and contain:
  - grant codeBase "file:<absolute_path>/PolicyFiles/lib/**" {
      permission java.io.FilePermission "<absolute_path>/PolicyFiles/
      Data/**", "read";
    }
  - This policy file grants all code that is loaded from the lib directory with
    the file permission to read files in the directory Data

• Execute PrintKeyStore specifying security manager and policy file from the
  lib directory:
  - java -Djava.security.manager -Djava.security.policy=policy
    -jar lib/PrintKeyStore.jar Data/timStore timword
7.d.8 Controlling Access to Runtime System Resources

- Resources that have predefined Permissions include:
  - **local file system** - read, write, execute, delete; see: `java.io.FilePermission`
  - **sockets** - listen, connect, accept, resolve
    - e.g. `permission java.net.SocketPermission "localhost:1024-", "listen";`
    - allows listening to the un-privileged ports
  - **reflect** - suppress checks to public, package, protected and private
    - see: `java.lang.reflect.ReflectPermission`
  - **security** - control access to security objects
    - see: `java.security.SecurityPermission`
    - e.g.: `perm - new java.io.FilePermission("${user.home}/-", "read");`
    - example of security object created by execution of Java virtual machine
  - **runtime** - control of classloading, security manager and threads
    - see: `java.lang.RuntimePermission`
  - **property** - read, write; see: `java.lang.System.getProperties()`, and the property permission: `java.util.PropertyPermission`
7.d.9 Specifying Policy Files and Common Permissions

- Policy Files may include optional codeBase and signedBy and keyStore
  - See the `j2sdk` javadocs information on `policytool`
  - If you specify `signedBy` then you must also include a `keyStore`
  - Where a file pathname is required, it must be system specific (unix vs. windows), but where a URL is required, it must contain only slashes (`/`).

- When specifying `codeBase` in the policy file a URL is used, such as:
  - `grant codeBase "file:/${user.dir}/classes/-" {...}`
  - `dash at end (-)` matches all class and jar files recursively to subdirectories
  - `star at end (*)` matches all class and jar files in the specified directory
  - `ending with / only` specifies all class files (no jars) in specified directory

- `java.io.FilePermission`
  - Backslash is an escape character. Filenames entered in policytool can have single \\ (backslash). In policy files must be double (c:\\ser321\\tmp).
  - `dash (-) and star (*)` at the end have same meanings as with codeBase
  - On unix: `java.io.FilePermission "${user.dir}/ServerData/-", "read"`;
  - windows: `java.io.FilePermission "${user.dir}\ServerData\\-","read"`;
7.e Cryptography

7.e.1 The Java Cryptography Architecture

- From the Java tutorial, the JCE includes a provider architecture that allows for multiple and interoperable cryptography implementations.
7.e.2 Using Crypto Services - Example

• Currently, high strength cryptographic services are included in the java runtime environment, but are limited by export controls. Thus the default policy files with jdk downloads restricts strength. A separate download is necessary for unlimited strength, but the associated policy files are limited by region/country. See: **JCE Unlimited Strength Policy Files**
7.e.3 The Multiple Provider Approach

- An engine class abstractly defines a cryptographic service (without a concrete implementation).

  - Signature, MessageDigest, and KeyPairGenerator (in jdk1.1)
  - CertificateFactory - This class defines the functionality of a certificate factory, which is used to generate certificate and certificate revocation list (CRL) objects from their encodings.
  - KeyStore class (an engine class) that supplies well-defined interfaces to access and modify a repository of keys and certificates

- Example of transparency to application developer
  - An API client may request and use an instance of the Signature engine class to access the functionality of a digital signature algorithm to digitally sign a file.
  - The actual implementation supplied in a SignatureSpi subclass would be for a specific kind of signature algorithm, such as SHA-1 with DSA or MD5 with RSA.
7.e.4 The Multiple Provider Approach

- Java 2 security model allows multiple providers to supply:
  - Digital Signature implementation
  - Keystore creation and management
  - Algorithm parameter creation and management
  - Key factory support to convert between different key representations
  - Certificate factory support to manage certificates

- Security objects are not created with a constructor.
  - public static Signature getInstance(String algorithm)
  - getInstance handles mediation among SPI CSP providers.

- After creation, security objects must be initialized using init methods
7.e.5 Ciphers

- The Java Cryptology Architecture (JCA) supports encryption and decryption.
- **Encryption** is the process of taking data (called clear text) and a short key and producing cipher data that is meaningless to anyone who does not know the key.
- **Decryption** is the process of taking cipher text and a short key to produce the corresponding clear text.
- A **Cipher** is an object capable of carrying out encryption and decryption according to an encryption algorithm.
- The **JCA** provides a framework and implementations for encryption, key generation, and **Message Authentication Code (MAC)** algorithms.
- The architecture supports implementation independence to support multiple suppliers
- To the extent possible it supports Cipher algorithm independence.
7.e.6 Ciphers

- Constructing Cipher objects
  - Cipher c1 = Cipher.getInstance("DES/EBC/PKCS5Padding");
  - string format is: algorithm / mode / padding

- After a Cipher has been initialized, c1.init(...)
  - encryption or decryption is specified in call to init

- Encryption and decryption
  - Single-part by a single call to doFinal(inBuf);
  - Multi-part: several update(inBuf), and one doFinal(inBuf);

- See example encryption program CipherTest.java, which is contained in the project: cryptography.jar
  - javac -d classes CipherTest.java
  - java -cp classes ser321.security.CipherTest
7.e.7 Keys

- Key management varies based on usage, algorithm and supplier.
- **Algorithm independent** keys
  - `KeyGenerator kg = KeyGenerator.getInstance(algo, provider);`  
    `kg.init(keySize, random);`
- **Algorithm specific** keys
  - `KeyGenerator kg = KeyGenerator.getInstance(algo, provider);`  
    `kg.init(algorithmParamSpec, random);`
- **Symmetric** keys - the same key used for encryption is also used for decryption
- **Key pairs** - Private key is used for encryption, public key used for decryption
- **Password Based Encryption (PBE)** - Use the same password for decryption as was used for encryption.
7.e.8 Password Based Encryption/Decryption

- See Mohan Atreya’s e-security write-ups at:
  - Introduction to Cryptography. See: [https://web.cs.ship.edu/~cdgira/courses/CSC434/Fall2004/docs/course_docs/IntroToCrypto.pdf](https://web.cs.ship.edu/~cdgira/courses/CSC434/Fall2004/docs/course_docs/IntroToCrypto.pdf)
  - Password Based Encryption.
  See: [https://pdfs.semanticscholar.org/72b4/ae505f15127507388c45cbad6bcafe6271c.pdf](https://pdfs.semanticscholar.org/72b4/ae505f15127507388c45cbad6bcafe6271c.pdf)

- Password based encryption/decryption builds the key based on a password.
- Sender and receiver must both know and use the same password.
- Example `PbecipherFile.java` which is contained in the project: `cryptography.jar`
7.e.9  Keys: Asymmetric versus Symmetric

- Both approaches have strengths and weaknesses
- **Asymmetric** = Private Key, Public Key pair
  - Key exchange of Public Keys is a good approach
  - Computations (both encrypt and decrypt) are very slow
  - Anyone with the public key can understand all messages sent
  - When used with a certificate, public keys can provide appropriate assurances for trust
- **Symmetric** = Same key is used on both ends of the communication
  - Data can be encrypted and decrypted quickly
  - Logistics of exchanging key can be a problem
7.e.10 Sealed Objects

- **javax.crypto.SealedObject** Class: Any serializable object can be encrypted/decrypted as a Sealed Object

- **Sealing the object**

  ```java
  // create Cipher object
  byte[] sKey; // the secret DES key (already generated)
  Cipher c = Cipher.getInstance("DES");
  c.init(Cipher.ENCRYPT_MODE, sKey);
  // do the sealing
  SealedObject so = new SealedObject("This is a secret", c);
  dc.init(Cipher.DECRYPT_MODE, sKey);
  // One way to unseal the object is to pass the cipher (dc) with the object (so).
  dc transparently includes the key
  try {
      String s = (String)so.getObject(dc);
  } catch (Exception e) {
      // do something
  }
  ```
7.f Secure Sockets with Java

7.f.1 The Java Secure Sockets Extension (JSSE)

- Java Secure Sockets address three security concerns for socket-based client-server applications
  - **Authentication** - to be certain who I am communicating with
  - **Integrity** - to be certain the information is not altered in communication
  - **Cryptography** - to be certain no one else can intercept and read the communications

- JSSE communication begins with an SSL handshake
  - Negotiate the cipher suite - which algorithm and parameters to use
  - Authenticates identity - establish secret key by client encrypting with server’s public key
  - Establish agreement on encryption mechanisms

- For further information, see the javadocs from j2sdk1.4(+)
  - `<java_home>/docs/technotes/guides/security/index.html`
7.f.2 SSL Messages

SSL Messages

Client

1. Client hello

7. Certificate optional
8. Client key exchange
9. Certificate verify optional
10. Change cipher spec
11. Finished
14. Encrypted data

Server

2. Server hello
3. Certificate optional
4. Certificate request optional
5. Server key exchange optional
6. Server hello done
12. Change cipher spec
13. Finished
14. Encrypted data
7.f.3 Example Client-Server using JSSE (Secure Sockets)

- Java code for the server: see the package javax.net.ssl

  - SSLSockServer.java See the example project: secureSockets.jar

  SSLServerSocket sslServer;
  SSLSocket sslSock;
  try {
    SSLServerSocketFactory sslServFact =
    (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
    sslServer = (SSLServerSocket)sslServFact.createServerSocket(8888);
    sslSock = (SSLSocket)sslServer.accept(); ...
  }

- Java code for client

  - SSLSockClient.java

  SSLSocket sslSock = null;
  try {
    OutputStream out;
    SSLSocketFactory sslFact
    =(SSLSocketFactory)SSLSocketFactory.getDefault();
    sslSock = (SSLSocket)sslFact.createSocket("localhost", 8888);
    out = sslSock.getOutputStream();
  }...

7.f.4 Setting up Keys for JSSE

- First, set up the key pair to be
  - keytool -genkey -alias tim -keyalg RSA -validity 7 -keystore timStore
  - keytool -genkey -alias nick -keyalg RSA -validity 7 -keystore nickStore
  - generate the key with the same password as the store (timpwd and nickpwd in this example)

- Check that the keystore includes a private key
  - keytool -list -v -keystore timStore

- Export a certificate (public key) for tim and import it into nickStore
  - keytool -export -alias tim -keystore timStore -rfc -file tim.cer
  - keytool -import -alias timCert -file tim.cer -keystore nickStore
  - keytool -list -v -keystore nickStore
7.f.5 Example Client-Server using JSSE (Secure Sockets)

- See the example project: `secureSockets.jar`
- Compile the client and server
  
  - `javac -d classes/ src/SSLSockServer.java src/SSLSockClient.java`
- Execute the server in one window specifying the **private key entry**
  
  - `java -cp classes -Djavax.net.ssl.keyStore=timStore
  -Djavax.net.ssl.keyStorePassword=timpwd
  -Djavax.net.ssl.keyStoreType=JKS
  ser321.security.SSLSockServer`
- Execute the client in a different window specifying **public key certificate**
  
  - `java -cp classes -Djavax.net.ssl.trustStore=nickStore
  -Djavax.net.ssl.trustStorePassword=nickpwd
  -Djavax.net.ssl.trustStoreType=JKS
  ser321.security.SSLSockClient`