Java Security API

Slides for AISS course (Prof. Ricardo Chaves),
slightly adapted in 2017/18 for SEC

Platform Security

- **The Java™ platform was designed with a strong emphasis on security**
- **Core language features:**
  - Strong data typing
  - Automatic memory management
  - Garbage collection
  - Range-checking on arrays
  - Access modifiers (public, protected, private)
  - Byte-code verification
  - Secure class loading

Java Security Technology

- **Java security technology includes a large set of APIs, tools, and implementations of commonly used security algorithms, mechanisms, and protocols:**
  - cryptography
  - public key infrastructure
  - secure communication
  - authentication
  - access control

Basic Security Architecture

- **Security APIs were designed around the following principles**
  - Implementation independence
    - Applications request generic security services from the Java platform via **providers**
  - Implementation interoperability
    - Providers are interoperable across applications
  - Algorithm extensibility
    - The Java platform includes a number of built-in providers, and supports the installation of custom providers

Java Security API packages

- **Java Cryptography Architecture (JCA) is part of Java 2 run-time environment.**
  → <u>java.security.*</u>
  - *java.security* package includes classes used for authentication and digital signature
- **JCE adds <u>encryption and decryption APIs</u> to JCA.**
  → <u>javax.crypto.*</u>
  - *javax.crypto* package contains Java Cryptography Extension classes

Java Cryptography Extension

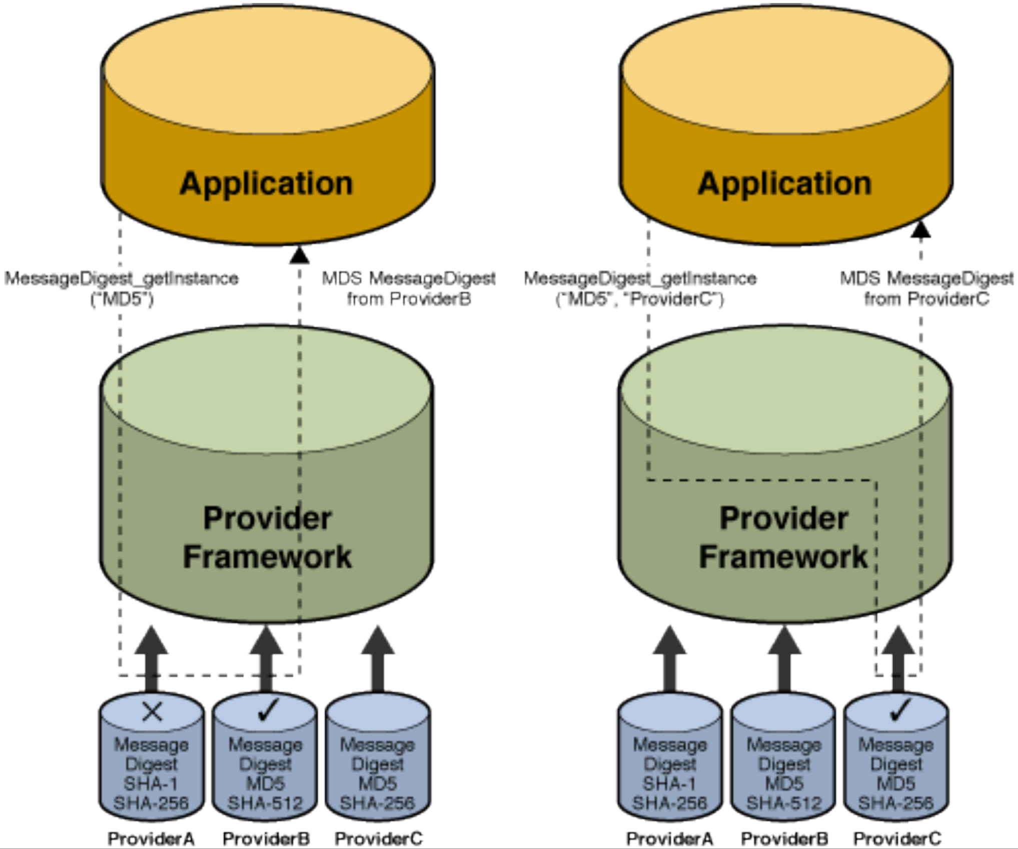
- **The JCE allows different implementations from many providers, by defining different types of cryptographic "engines" (services)**
- **An engine class provides the interface to a specific type of cryptographic service, independent of a particular cryptographic algorithm or provider**
- **Useful classes are:**
  - SecretKeyFactory
  - Cipher
  - SealedObject
  - KeyGenerator
  - KeyAgreement
  - Mac
  - SecureRandom

Requesting a service

- **To use the JCA, an application:**
  - requests a particular type of object (such as a **`MessageDigest`**)
  - and a particular algorithm or service (such as the "MD5" algorithm)
  - and gets an implementation from one of the installed providers

```
try {
    MessageDigest md = MessageDigest.getInstance("MD5");
}
catch (NoSuchAlgorithmException e) {
    // no such algorithm provided
}
```

Provider selection




```
md = MessageDigest.getInstance("MD5");   /* default provider */
md = MessageDigest.getInstance("MD5", "ProviderC");
```

Security Providers

- **Implementation independence is achieved using a "provider"-based architecture**
- **Provider - a package or set of packages that implement one or more security services**

```
import java.security.*;

Provider[] providers =
        Security.getProviders();
for (Provider p: providers){
   System.out.println(p.toString());
}
```

Java 6.0 →

SUN version 1.6
SunRsaSign version 1.5
SunJSSE version 1.6
SunJCE version 1.6
SunJGSS version 1.0
SunSASL version 1.5
XMLDSig version 1.0
SunPCSC version 1.6
SunMSCAPI version 1.6

JCE Providers

- **Open source providers: Cryptix and Bouncy Castle**
- **Plugged-in by:**
  - modifying the *java.security* file
  - using code to add a provider (dynamically)

Example:

import cryptix.jce.provider.CryptixCrypto;

Provider cryptix_provider = **new** CryptixCrypto();

int result=Security.addProvider(cryptix_provider);

Listing provider services

```
Provider[] providers = Security.getProviders();
for (Provider p: providers){
   System.out.println(p.toString());
   Set<Service> services = p.getServices();
   for (Service s: services){
      System.out.println("     " + s.getType() +
                                  " --> " + s.getAlgorithm());
   }
}
```

SUN version 1.6 services

SUN version 1.6	AlgorithmParameterGenerator --> DSA
SecureRandom --> SHA1PRNG	AlgorithmParameters --> DSA
Signature --> SHA1withDSA	KeyFactory --> DSA
Signature --> NONEwithDSA	CertificateFactory --> X.509
KeyPairGenerator --> DSA	KeyStore --> JKS
MessageDigest --> MD2	KeyStore --> CaseExactJKS
MessageDigest --> MD5	Policy --> JavaPolicy
MessageDigest --> SHA	Configuration --> JavaLoginConfig
MessageDigest --> SHA-256	CertPathBuilder --> PKIX
MessageDigest --> SHA-384	CertPathValidator --> PKIX
MessageDigest --> SHA-512	CertStore --> LDAP
	CertStore --> Collection
	CertStore --> com.sun.security.IndexedCollection

The `SecureRandom` Class

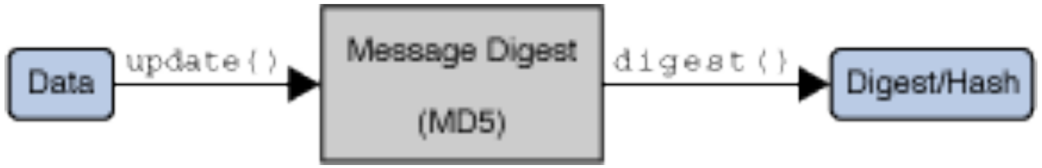
- **Provides the functionality of a Random Number Generator**
- **Produces cryptographically strong random numbers**

```
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
System.out.println("Int: " + random.nextInt());
System.out.println("Float: " + random.nextFloat());
System.out.println("Long: " + random.nextLong());
System.out.println("Boolean: " + random.nextBoolean());
```

```
Int: 256421598
Float: 0.63456607
Long: 7589616350181670704
Boolean: true
```

The `MessageDigest` Class

- **Designed to provide the functionality of cryptographically secure message digests such as SHA-1 or MD5**



- **The MD5 algorithm produces a 16 byte digest, and SHA-1's is 20 bytes**

Computing a **MessageDigest** object

```
MessageDigest sha = MessageDigest.getInstance("SHA-1");
byte[] i1 = "Hello World".getBytes();
sha.update(i1);
byte[] hash = sha.digest();
System.out.println((new BASE64Encoder()).encode(hash));

byte[] i2 = "Hello World!".getBytes();
sha.update(i2);
hash = sha.digest();
System.out.println((new BASE64Encoder()).encode(hash));
*********************************************************************************************************************************

sha.update(received);
hash = sha.digest();
System.out.println((new BASE64Encoder()).encode(hash));
```

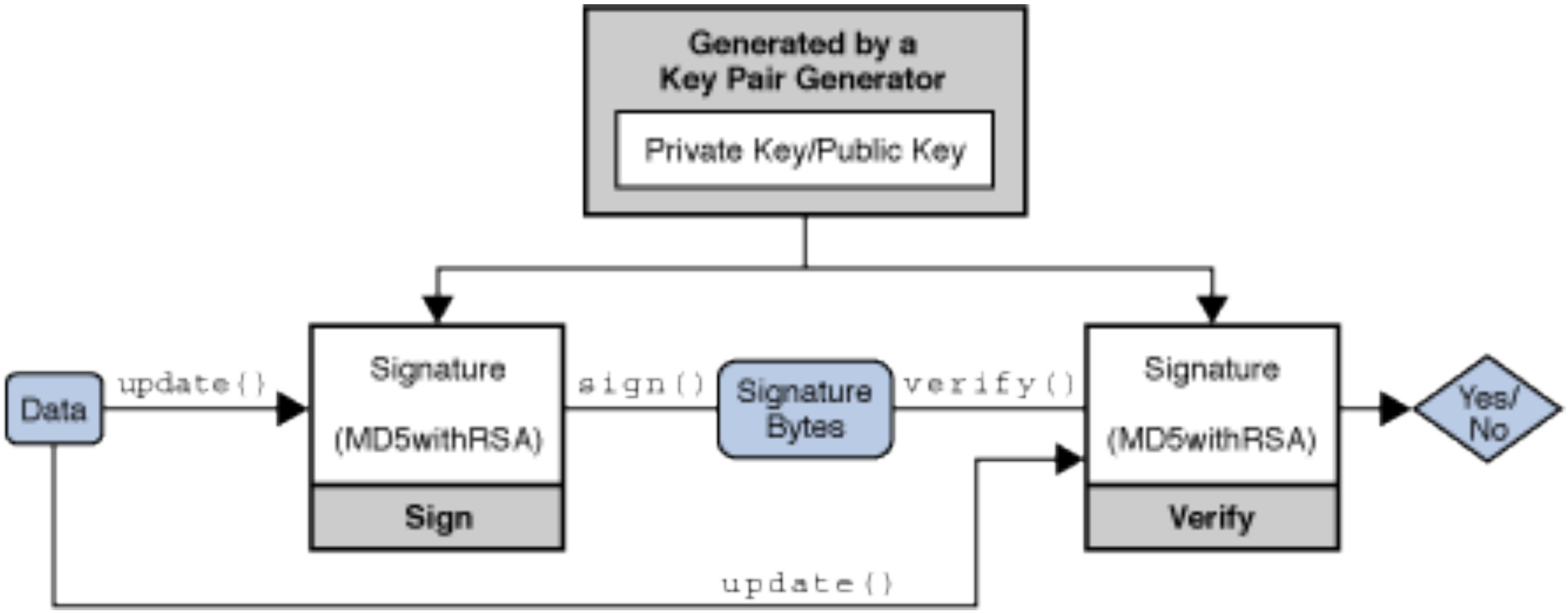
```
Ck1VqNd45QIvq3AZd8XYQLvEhtA=

Lve95gjOVATpfV8EL5X4nxwjKHE=

Ck1VqNd45QIvq3AZd8XYQLvEhtA=
```

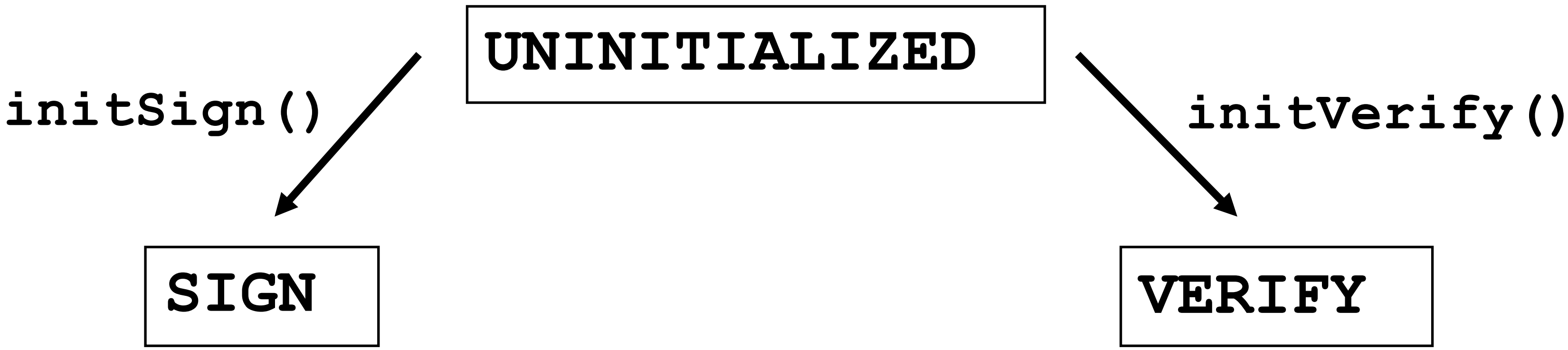
The `Signature` Class

- **Provide the functionality of a cryptographic digital signature algorithm such as DSA**



`Signature` Object States

- **`Signature` object is always in a given state, where it may only do one type of operation**
- **The three states a `Signature` object may have are:**



Generating a Pair of public/private Keys

- **First step is to generate public/private key pair**
- **All key pair generators share the concepts of a keysize and a source of randomness**

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");

SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
keyGen.initialize(1024, random);
KeyPair pair = keyGen.generateKeyPair();

PrivateKey privateKey = pair.getPrivate();
PublicKey publicKey = pair.getPublic();
```

Generating/verifying a signature

```java
byte[] data = "Data to be signed".getBytes();
// generating a signature
Signature dsaForSign = Signature.getInstance("SHA1withDSA");
dsaForSign.initSign(privateKey);
dsaForSign.update(data);
byte[] signature = dsaForSign.sign();
```

```java
// verifying a signature
Signature dsaForVerify = Signature.getInstance("SHA1withDSA");
dsaForVerify.initVerify(publicKey);
dsaForVerify.update(data);
boolean verifies = dsaForVerify.verify(signature);
System.out.println("Signature verifies: " + verifies);
```

The `Cipher` Class

- A cryptographic cipher for encryption and decryption can be instantiated using the Cipher.getInstance factory method
- Associated with a transformation name in the format, *algorithm/mode/padding*
- Can operate within four modes: encrypt, decrypt, key wrap, key unwrap.
- Must be initialized using a specified mode, and secret key information.

The `Cipher` Class

- **Methods:**
  - getInstance(String algorithm)
    - Generates a Cipher object that implements the specified algorithm.
  - init(int opmode, Key key)
    - The cipher is initialized with a key for either encryption or decryption.
  - doFinal(byte[] input)
    - Encrypts or decrypts data in a single-part operation, or finishes a multiple-part operation, depending on how this cipher was initialized.
  - update(byte[] input)
    - Continues a multiple-part encryption or decryption operation.

The `Cipher` Class

- **Class:Javax.crypto.Cipher**
  - Available algorithms:

```
for (String a: Security.getAlgorithms("Cipher")){
      System.out.println(a);
}
```

ARCFOUR	DESEDE
PBEWITHMD5ANDDES	AESWRAP
RC2	AES
RSA	DES
PBEWITHMD5ANDTRIPLEDES	DESEDEWRAP
PBEWITHSHA1ANDDESEDE	RSA/ECB/PKCS1PADDING
	PBEWITHSHA1ANDRC2_40

Using Encryption (AES)

```
// Generate AES key
KeyGenerator keygen = KeyGenerator.getInstance("AES");
keygen.init(128); // initialize the key size
SecretKey aesKey = keygen.generateKey();

// Initialize cipher object
Cipher aesCipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
aesCipher.init(Cipher.ENCRYPT_MODE, aesKey);

byte[] cleartext = "Data to be encoded".getBytes();

// Encrypt the cleartext
byte[] ciphertext = aesCipher.doFinal(cleartext);

// Initialize the same cipher for decryption
aesCipher.init(Cipher.DECRYPT_MODE, aesKey);

// Decrypt the ciphertext
byte[] cleartext1 = aesCipher.doFinal(ciphertext);
```

Encryption Exceptions

```
try {
        // algorithm from previous slide
        . . .
        System.out.println("Cipher successful!");
}
catch (NoSuchAlgorithmException e1) {. . .}

catch (NoSuchPaddingException e2) {. . .}

catch (BadPaddingException e3) {. . .}

catch (InvalidKeyException e4) {. . .}

catch (IllegalBlockSizeException e5) {. . .}
```

JCA - Secure Key Storage

- **Keys need to be stored on secondary storage so that programs can access them conveniently and securely for subsequent use.**
- **JCA provides an extensible architecture to manage keys through KeyStore.**
- **A KeyStore object maintains an in-memory table of key and certificate entries, indexed by alias strings, allowing retrieval, insertion and deletion of entries.**
- **Keystore files are usually password protected.**

Class: java.security.KeyStore

- **Methods:**
  - getInstance (String type)
    - Create an instance of KeyStore of the specified type.
  - load(InputStream stream, char[] password))
    - Open keystore with password and load keys from keystore file to memory
  - getKey(String alias, char[] password)
    - Access the keystore with password and get the key based on a given key alias
  - setEntry(String alias, KeyStore.Entry entry, KeyStore.ProtectionParameter protParam)
    - Set a new key entry in the keystore
  - store(OutputStream stream, char[] password)
    - Store this keystore to the given output stream, and protect its integrity with the given password.

Example: Create an empty KeyStore object

- **The following sample creates an empty KeyStore object with password protection.**

```
// Create an instance of KeyStore of type "JCEKS".
// JCEKS refers the KeyStore implementation from SunJCE provider
ks = KeyStore.getInstance("JCEKS");

// Load the null Keystore and set the password to "changeme"
ks.load(null, "changeme".toCharArray());
```

Example:Set Key Entry

- **The following sample sets the generated key "mykey" in the KeyStore.**

```
//Create an instance of KeyStore.SecretKeyEntry using "mykey"
KeyStore.SecretKeyEntry skEntry = new KeyStore.SecretKeyEntry(mykey);

//Get key alias name from user input.
String alias=args[0];

//Create KeyStore Password
KeyStore.PasswordProtection password;
password = new KeyStore.PasswordProtection("changeme".toCharArray());

//Set the key entry in the key store with an alias.
ks.setEntry(alias, skEntry, password);
```

Example:Store KeyStore object in file

- **The following sample writes the KeyStore object into a file for storage.**

```
//Create a new file to store the KeyStore object
java.io.FileOutputStream fos = new
    java.io.FileOutputStream("keystorefile.jce");

//Write the KeyStore into the file
ks.store(fos, "changeme".toCharArray());

//Close the file stream
fos.close();
```

Example:Retrieving Keys from KeyStore

- **The following sample retrieves keys from a KeyStore file.**

```
//Open the KeyStore file
FileInputStream fis = new FileInputStream("keystorefile.jce");

//Create an instance of KeyStore of type "JCEKS"
ks = KeyStore.getInstance("JCEKS");

//Load the key entries from the file into the KeyStore object.
ks.load(fis, "changeme".toCharArray());
fis.close();

//Get the key with the given alias.
String alias=args[0];
Key k = ks.getKey(alias, "changeme".toCharArray());
```

JCE - SealedObject

- **For securely persisting objects that can be serialized.**
- **Instantiated with a Cipher object and a serializeable object.**
- **Any algorithm parameters used by the Cipher object are stored in the SealedObject for easy decryption.**

JCE - KeyAgreement

- Lets Alice and Bob establish a secret key in an insecure environment.
- Uses an asymmetric system. A developer must choose the key agreement algorithm. (e.g., Diffie-Hellman)
- The 'generateSecret' method returns the established secret key
- The 'doPhase' method performs the exchange
- Example:

```
KeyAgreement ka = KeyAgreement.getInstance("DH");
ka..init( alicePrivateKey );
ka..doPhase( bobPublicKey, true );
byte[] secret = ka.generateSecret();
```

Authentication in Java

Definitions

- **Authentication is the process of determining the identity of a user**
- **Authorization is the process of giving user permission to do or to have something**
- **Logically, authorization is preceded by authentication**

JAAS

- **Java™ Authentication and Authorization Service: Authentication and user-based access control services in Java**
- **JAAS can be used for two purposes:**
  - for the authentication of users
    - to reliably and securely determine who is currently executing Java code
  - for the authorization of users
    - to ensure they have the access control rights (permissions) required to do the actions performed

Generating MACs in Java

- **Sequence of Steps:**
  - Create a KeyGenerator for HmacMD5
  - Generate the shared secret
  - Create a MAC object, initialize it with shared secret (init method)
  - Pass byte array to "doFinal" method of MAC

Generating MACs (example)

```
KeyGenerator keygen = KeyGenerator.getInstance("HmacMD5");
SecretKey sk = keygen.generateKey();
Mac authenticator = Mac.getInstance(sk.getAlgorithm());
authenticator.init(sk);
byte[] msg = "Hello World".getBytes();
byte[] msgAuthenticator = authenticator.doFinal(msg);
```

Acknowledgments

- Some of these slides are based on material by Prof. Douglas Lyon, Fairfield University, http://http://www.docjava.com/