Basic Cryptography

- Introduction
- Cryptographic Building Blocks
- Key Management Issues
- Software interfaces to cryptographic primitives

1

Introduction

- Definition
  - *Cryptography* is the scientific study of mathematical techniques relating to **information security**
- Goals of cryptography:
  - message confidentiality (= privacy, secrecy)
  - message integrity
  - message or entity authentication
  - non repudiation

2

Cryptographic Primitives

- Introduction
- Cryptographic Building Blocks
- Key Management Issues
- Software interfaces to cryptographic primitives

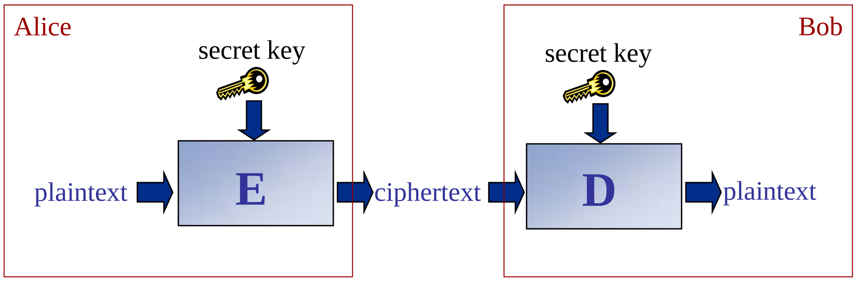
3

Cryptographic Building Blocks

- Symmetric cryptography
- Public-key cryptography
- Hash functions
  - Unkeyed hash functions
  - Message Authentication Codes (MAC's)
- Digital signatures
- Secure random numbers

4

Symmetric Cryptography



- NOTE: Algorithm secrecy ↔ key secrecy

5

Cryptanalytic Attacks

- Algorithm should be secure against
  - Ciphertext-only attack
    - Find *k* or plaintext given only ciphertext.
  - Known-plaintext attack
    - Find *k* given $\langle M_1, C_1\rangle, \langle M_2, C_2\rangle, \ldots$
  - Chosen-plaintext attack
    - Known-plaintext, but adversary chooses $M_1, M_2, \ldots$
  - Chosen-ciphertext
    - Known-plaintext, but adversary chooses $C_1, C_2, \ldots$
- Security depends on:
  - Algorithm: use well-known algorithms
  - Key-length: longer keys improve security

6

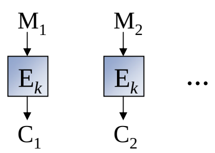
Block and stream ciphers

- Block ciphers encrypt fixed-size input blocks
  - *Padding* may be necessary.
  - Different *modes* of operation on arbitrary sized streams (see next slide)
  - Block size influences security of the cipher
- Stream ciphers can encrypt bit-by-bit
  - e.g. one-time-pad
  - Key stream generators

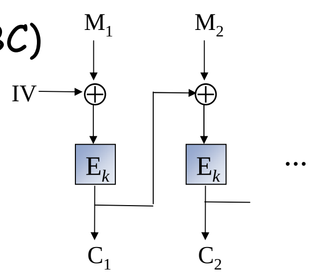
7

Encryption modes (block ciphers)

- Electronic Codebook (ECB)



- Cipher Block Chaining (CBC)



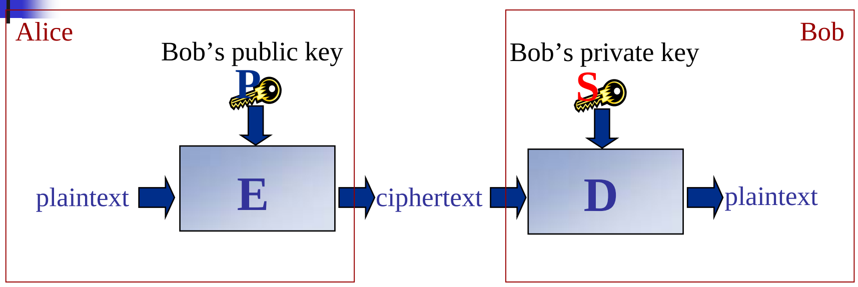
8

Real-world Algorithms

- DES (Data Encryption Standard)
  - Designed by IBM in 1970's, influenced by NSA
  - 64-bit blocks, 56-bit key (too short nowadays)
- Triple DES
  - Three DES encryptions with independent keys
- AES (Advanced Encryption Standard) / Rijndael
  - Made in Belgium
  - Variable key/block length (128, 192 or 256 bits)
- RC4
  - Proprietary stream cipher of RSA Labs

9

Public-key Cryptography



- Key generation algorithm
- Should be secure against the same attacks as symmetric encryption
- Easier key management but slower

10

Public-key Cryptography

- Public-key ciphers are all block ciphers
  - Block size is much larger than for symmetric ciphers
  - Typically only single block encryption to encrypt a symmetric key
  - Padding is more elaborate to deal with small message space attacks
    - *Randomization* of the plaintext

11

Real-world Algorithms

- RSA (Rivest, Shamir, Adleman)
  - Widely used: de facto standard for public-key cryptography
  - Variable key length
  - Based on problem of factoring large integers
- ECC (Elliptic Curve Cryptography)
  - For wireless and embedded environments
- Others exist but not frequently used
  - e.g. Rabin, ElGamal, ...
- Padding algoritms
  - PKCS#1 v1.5
  - OAEP

12

Notational Conventions

- Notation for keys:
  - Symmetric key: K, K_{AB}
  - A's public key: PK_A
  - A's private key: SK_A
- Notation for encryption:
  - ciphertext = {plaintext}K
  - ciphertext = {plaintext}PK

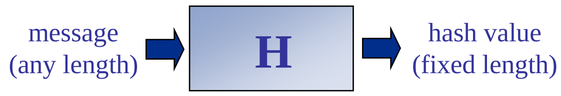
13

Hash Functions

- Definition
  - Maps arbitrary strings on fixed-length hash values
  - "Fingerprint" of message
  - AKA *Message Digest*
- Cryptographic hash functions are:
  - One way
  - Collision resistant
- Two flavors: keyed (MAC's) and unkeyed

14

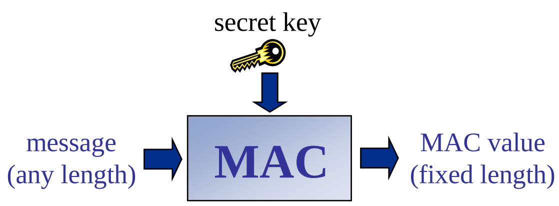
Unkeyed Hash Functions



- One way:
  - Easy to compute hash value for message
  - Hard to find message with specific hash value
- Collision resistant:
  - Hard to find second message with same hash value
- Used for detecting unauthorized changes
  - e.g. Detection of virus infection

15

Message Authentication Codes



- Properties:
  - One way
  - Collision resistant
  - Protected by secret key:
    - Computing and checking impossible without key
- Used for message integrity check

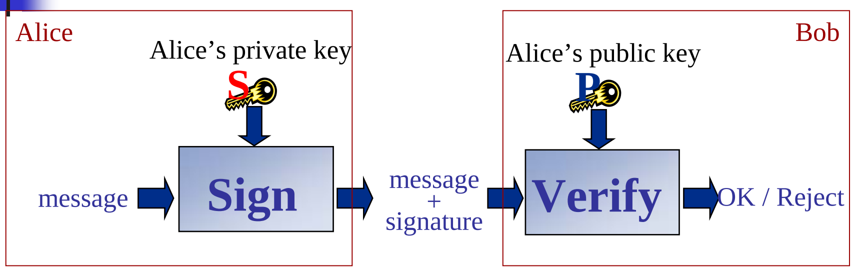
16

Real-world Algorithms

- Unkeyed hash functions:
  - SHA-1 (Secure Hash Algorithm)
    - Designed by NSA
    - Arbitrary-length input → 160-bit output
  - MD-5 (Message Digest)
    - By Ron Rivest
    - Arbitrary-length input → 128-bit output
- MAC's:
  - Any symmetric encryption of any hash function
  - Using only hash functions: $MAC_k(M) = H(k,M)$, or better: H-MAC turns any unkeyed hash in a MAC
  - DES-CBC-MAC: the last block of a CBC encryption

17

Digital Signatures



- Key generation algorithm
- Digital signatures provide:
  - Message origin authentication
  - Non repudiation

18

Digital Signatures

- Digital signatures also operate on fixed size input blocks
  - Padding is necessary but has completely different requirements than padding for encryption
    - E.g. no randomization
  - To sign arbitrary sized messages
    - Sign a hash of the message
- Standardized signature schemes specify how hashing and padding must be used

Real-world Algorithms

- RSA
  - Public key and private key are interchangeable
  - Signature = encryption with private key
  - Verification = decryption with public key
- DSA (Digital Signature Algorithm)
  - Designed by NSA
  - Key length from 512 to 1024 bits
- Elliptic curve variant of DSA (ECDSA)

Notational Conventions

- MAC's:
  - MAC value = [message]K
- Digital Signatures:
  - signature = [message]SK

Secure Random Numbers

- True randomness is slow to obtain:
  - physical processes: noise diode, coin tosses, …
  - timing user interface events
- Solution: Pseudo-Random Generators
  - John von Neumann: "*Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin*"
  - generate many (seemingly) random numbers starting from one seed

Secure Random Numbers

- Importance of random number generation:
  - Generating cryptographic keys
  - Generating "challenges" in cryptographic protocols
- Cryptographically secure randomness
  - Passes all statistical tests of randomness
  - Impossible to predict next bit from previous output bits
- Do not use a built-in random generator that uses an unknown algorithm!

Conclusions

- Designing cryptographic primitives is *extremely hard*
  - never try to design your own algorithms, use well-known algorithms
- Implementing cryptographic primitives is *extremely hard*
  - whenever possible, use a crypto library or API from a reputable vendor

Cryptographic Primitives

- Introduction
- Cryptographic Building Blocks
- Key Management Issues
  - Generating keys
  - Key length
  - Storing keys
  - Key establishment
- Software interfaces to cryptographic primitives

Generating Keys

- Algorithm security = key secrecy
- Key should be hard or impossible to guess
  - Human password → dictionary attack!
  - Better: hash of entire pass-phrase
  - Machine-generated → use cryptographically secure pseudo-random generator

Key Length

- Trade-off: information value ↔ cracking cost
- Symmetric algorithms
  - $1 000 000 investment in VLSI-implementation

56 bits	64 bits	128 bits
1 hour	10 days	10^{17} years

- Public-key algorithms

Year	*vs. Individual*	*vs. Corporation*	*vs. Government*
2000	1024	1280	1536
2005	1280	1536	2048
2010	1280	1536	2048

Storing Keys

- Simplest: human memory
  - Remember key itself
  - Key generated from pass-phrase
- Use Operating System access control
- Key embedded in chip on smart card
- Storage in encrypted form
  - *Key encryption* keys ↔ *data encryption* keys
- Limit key lifetime depending on
  - Value of the data
  - Amount of encrypted data

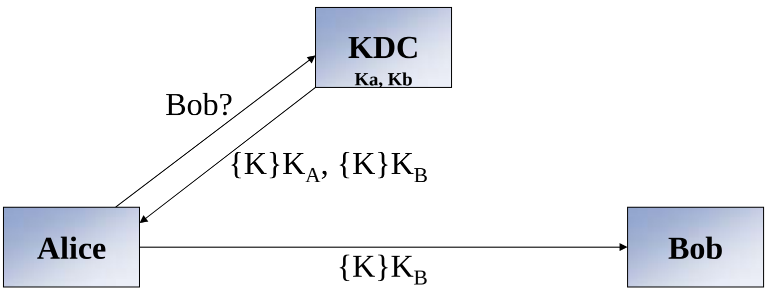
Key Establishment

- Key agreement = Two parties compute a secret key together
  - E.g. Diffie – Hellman protocol
- Key distribution or transport = One party generates a key and distributes it in a secure way to all authorized parties

29

Key Distribution

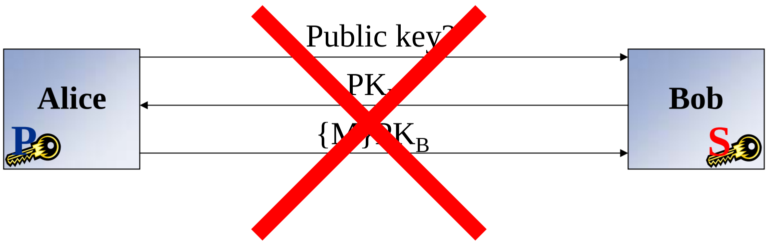
- Using symmetric encryption
  - Trusted party: Key Distribution Center (KDC)
  - General idea (oversimplified:)



30

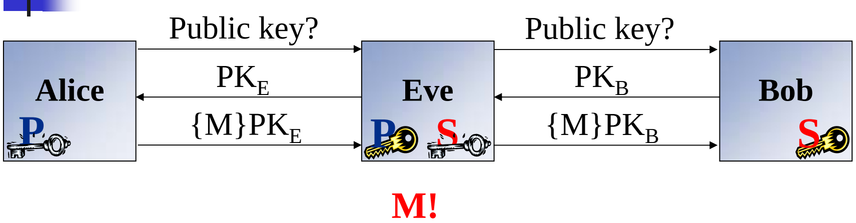
Key Distribution

- Using public-key encryption
  - No need for KDC?



– Man-in-the-middle attack!

Man-in-the-middle attack



- How can Alice be sure she got Bob's public key?
  - Solution: Certificates
    Public Key Infrastructure (PKI)

Cryptographic Primitives

- Introduction
- Cryptographic Building Blocks
- Key Management Issues
- Software interfaces to cryptographic primitives

33

Overview

- Design principles of modern API’s: Cryptographic Service Providers (CSP’s)
- The Java Cryptography Architecture and Extensions (JCA/JCE)
- The .NET cryptographic library

34

Design principles

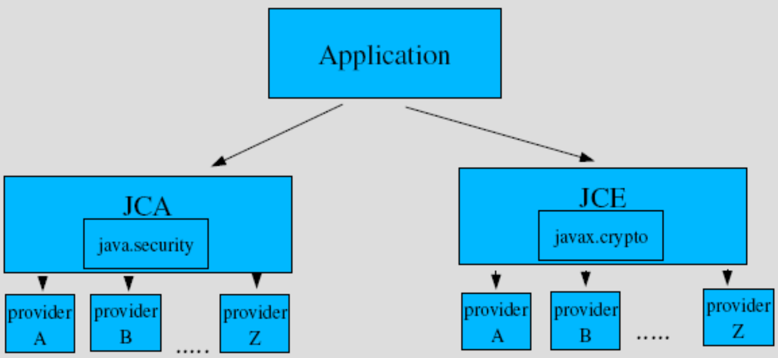
- Algorithm independence
  - *Engine* classes
- Implementation independence
  - *Provider* based architecture
- Implementation interoperability
  - *Transparent* and *opaque* data types

Bottom line: security mechanisms should be easy to change over time

35

Basic Architecture

- Provider based architecture



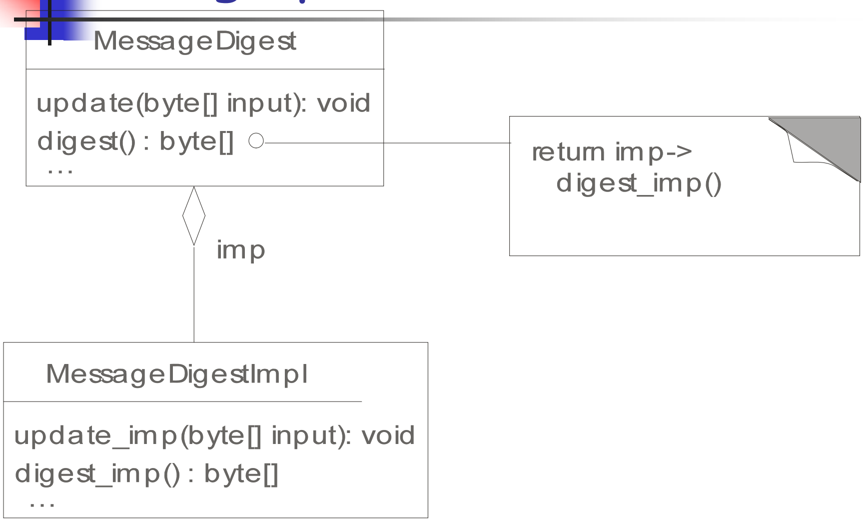
36

Engine classes

- Abstraction for a cryptographic service
  - interface between JCA and the actual implementation of the service classes
  - Provide cryptographic operations
  - Generate/supply cryptographic material
  - Generate objects encapsulating cryptographic keys
- Define the Cryptographic API
- Bridge pattern or inheritance hierarchy to allow for implementation independence
- Instances created by factory method

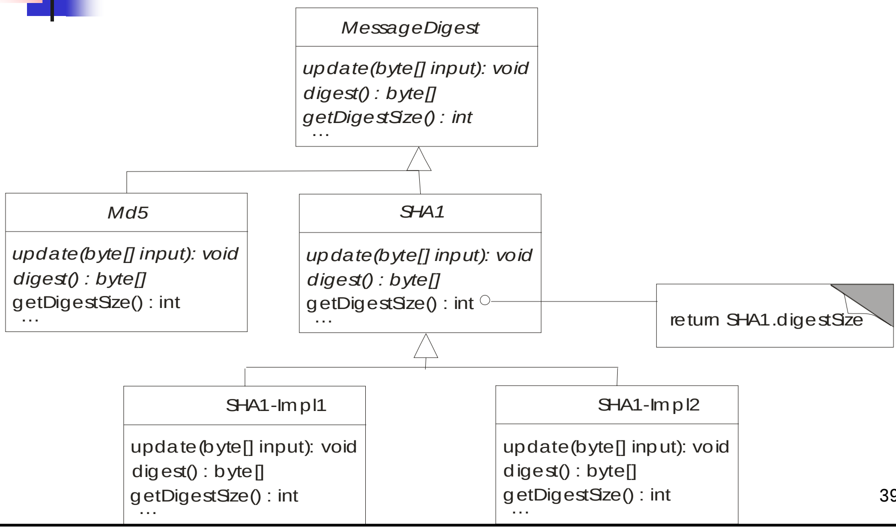
37

Bridge pattern



38

Inheritance based decoupling



39

Opaque vs transparent data

- Representation of data items like keys, algorithm parameters, initialization vectors:
  - Opaque: chosen by the implementation object
  - Transparent: chosen by the designer of the cryptographic API
- Transparent data allow for implementation interoperability
- Opaque data allow for efficiency or hardware implementation

40

Crypto frameworks and CSP's

- A *cryptographic framework* defines:
  - Engine classes (and possibly algorithm classes)
  - Transparent key and parameter classes
  - Interfaces for opaque keys and parameters
- A *cryptographic service provider* defines:
  - Implementation classes
  - Opaque key and parameter classes
  - Possibly methods to convert between opaque and transparent data

41

Example

- JCA implements a class, for example message digest,
- We know what a message digest is, but just having a generic message digest does not tell us anything
- The cryptographic service provider implements the actual algorithm, such as MD5 or SHA-1
- JCA implements the generic class
- The service provider implements the actual algorithm or type of cryptographic service that will be used

42

The JCA/JCE

- Java Crypto API structured as a cryptographic framework with CSP's
- Split in:
  - The *Java Cryptography Architecure (JCA)*
  - The *Java Cryptography Extensions (JCE)*
- This split is because of US export-control regulations for cryptography

43

US Export Restrictions

- US consider crypto software as munitions
  → export controls
  → no internal or import controls
- Before January 2000
  - Export of strong encryption products (> 40 bits) forbidden
    - Download is form of export!
  - No restrictions on authentication products
- Since January 2000: relaxed
  - Exception License needed for export
    - Received after technical review by NSA
  - Still forbidden to "Terrorist-7" countries

44

Engine classes (JCA)

java.security.*

- MessageDigest
  hash functions
- Signature
- SecureRandom
- KeyPairGenerator
  generate new key pairs
- KeyFactory
  convert existing keys
- CerticateFactory
  generate certificates from encoded form
- KeyStore
  database of keys
- AlgorithmParameters
- AlgorithmParameter-Generator

45

Engine classes (JCE)

javax.crypto.*

- Cipher
  encryption, decryption
- Mac
- KeyGenerator
  generate new symmetric keys
- SecretKeyFactory
  convert existing keys
- KeyAgreement

46

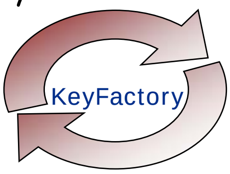
Key Classes

Opaque Representation

- No direct access to key material
- Encoded in provider-specific format
- java.security.Key

Transparent Representation

- Access each key material value individually
- Provider-independent format
- java.security.KeySpec



y = …
p = …
q = …
g = …

47

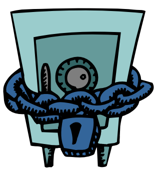
Parameter Classes

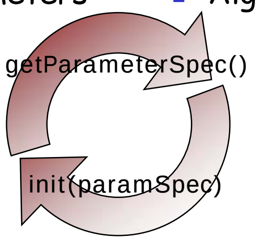
Opaque Representation

- No direct access to parameter fields
- Encoded in provider-specific format
- AlgorithmParameters

Transparent Representation

- Access each parameter value individually
- Provider-independent format
- AlgorithmParameterSpec





g = …
p = …
q = …

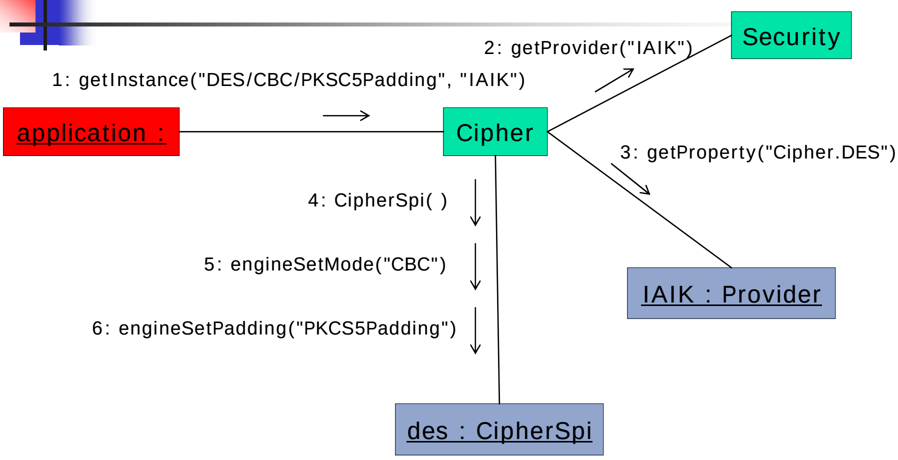
48

Overall structure of the framework

- Security class encapsulates configuration information (what providers are installed)
- Per provider, an instance of the provider class contains provider specific information (e.g. what algorithms are implemented in what classes)
- Factory method on the engine class interacts with the Security class and provider objects to instantiate a correct implementation object

49

Example: creating ciphers



50

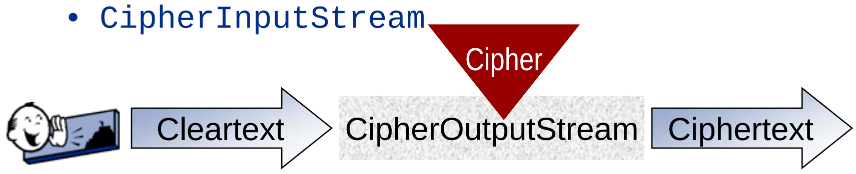
Additional support and convenience classes

- Secure streams
  - For easy bulk encryption and decryption
- Signed objects
  - Integrity checked serialized objects
- Sealed objects
  - Confidentiality protected serialized objects
- Working with certificates
- Keystores

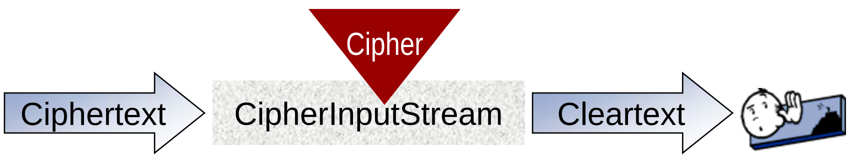
51

Secure Streams

- *Combination of Stream and Cipher object*
- `CipherInputStream`



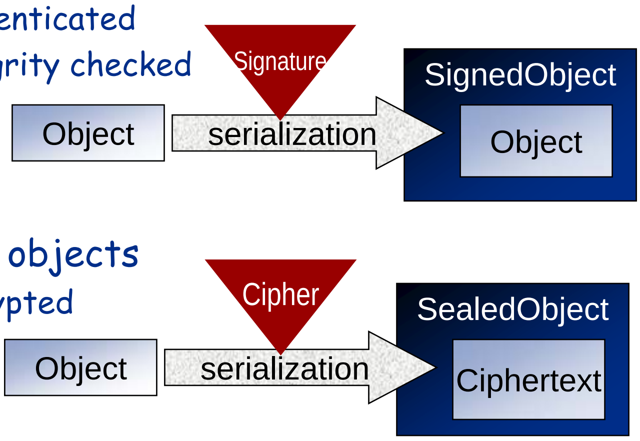
- `CipherOutputStream`



52

Secure Objects

- Signed objects
  - Authenticated
  - Integrity checked
- Sealed objects
  - Encrypted



53

Working with Certificates

- JCA/JCE does not have built-in support for generating new certificates
  - On purpose? (to make it harder for end-users to act as CA)
- Various commercial Java libraries implementing certificate generation on top of JCA/JCE are available
  - E.g. Baltimore KeyTools

54

Keystores

- Repository of
  - Secret keys (encrypted and integrity checked)
  - Private keys (encrypted and integrity checked)
  - Trusted certificates (integrity checked)
- KeyStore engine class
  - Access and modify keystore
  - Different *types*:
    - JKS: built-in default by Sun
      weak cryptography
    - JCEKS: included in JCE
      strong cryptography

55

JCA/JCE code examples

- Encryption
- Key factories and generation
- Digital signatures

56

Encryption Example

- Generate random session key

```
KeyGenerator keyGen =
      KeyGenerator.getInstance("DES", "SUN");
SecretKey sKey = keyGen.generateKey();
```

- Create and initialize cipher

```
Cipher cipher =
      Cipher.getInstance("DES/CBC/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, sKey);
```

57

Encryption example *(cont.)*

- Encrypt data (single stage)

```
cipherText = cipher.doFinal(clearText);
```

- Encrypt data (multi stage)

```
while ( <more bytes> ) {
    // produce clearText
    cipherText = cipher.update(clearText);   }
cipherText = c.doFinal();
```

58

Key Factory Example

- Create transparent key

```
BigInteger y = …;  BigInteger p = …;
BigInteger q = …;  BigInteger g = …;
DSAPublicKeySpec spec = new DSAPublicKeySpec(y, p, q, g);
```

- Convert to opaque key

```
KeyFactory kfac = KeyFactory.getInstance("DSA");
PublicKey dsaPubKey = kfac.generatePublic(spec);
```

- And back to transparent

```
PublicKeySpec spec2 =
    kfac.getKeySpec(dsaPubKey, DSAPublicKeySpec.class)
```


Key Pair Generator Example

- Create key pair generator

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

- Algorithm-independent initialization

```
keyGen.initialize(1024);
```

- Algorithm-specific initialization

```
p = …; q = …; g = …;
DSAParameterSpec dsaSpec = new DSAParameterSpec(p, q, g);
keyGen.initialize(dsaSpec);
```

- Generate key pair

```
KeyPair dsaPair = keyGen.generateKeyPair();
```

Signing and Verifying Example

- Create and initialize signature object

```
Signature signEngine = Signature.getInstance("SHA1withDSA");
PrivateKey priv = dsaPair.getPrivate();
signEngine.initSign(priv);
```

- Sign data

```
signEngine.update(data);
byte[] signature = signEngine.sign();
```

- Verify signature

```
PublicKey pub = dsaPair.getPublic();
signEngine.initVerify(pub);
signEngine.update(data);
boolean valid = signEngine.verify(signature);
```

61

Working with Certificates

- Reading in an encoded X.509 certificate:

```
CertificateFactory cf =
      CertificateFactory.getInstance("X.509");
X509Certificate cert =
      (X509Certificate)cf.generateCertificate(inStream);
inStream.close();
```

- Verifying a certificate:

```
cert.verify(publickey); // LIMITED verification!!!
```

- Accessing certificate information:

```
System.out.println(cert.getSubjectDN());
PublicKey pk = cert.getPublicKey();
```

62

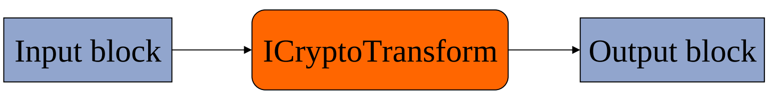
Overview

- Design principles of modern API's: Cryptographic Service Providers (CSP's)
- The Java Cryptography Architecture and Extensions (JCA/JCE)
- The .NET cryptographic library

63

The .NET cryptographic library

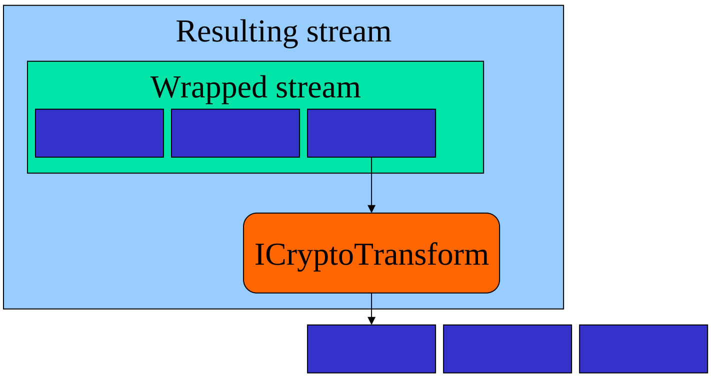
- CSP based library that uses inheritance based decoupling
- Bulk data processing algorithms are all made available as ICryptoTransforms
- Essentially 2 methods: TransformBlock() and TransformFinalBlock()



64

ICryptoTransform and CryptoStream

- ICryptoTransforms can wrap streams
  E.g. (in read mode)



65

Bulk data engine classes

- SymmetricAlgorithm, with algorithm classes
  - TripleDES, DES, Rijndael, …
- HashAlgorithm, with algorithm classes
  - SHA1, MD5, …
- KeyedHashAlgorithm, with algorithm classes
  - HMACSHA1, MACTripleDES, …

66

Asymmetric engine classes

- Generic AsymmetricAlgorithm engine class
  - RSA and DSA algorithm classes
- Specialized engine classes for typical uses of asymmetric cryptography, that take care of padding and formatting
  - AsymmetricKeyExchangeFormatter
  - AsymmetricSignatureFormatter
- In current version, asymmetric crypto is delegated to Windows CryptoAPI

67

Engine classes for key generation

- RandomNumberGenerator
  - For generating secure random numbers
- DeriveBytes
  - For deriving key material from passwords

68

Other functionality in the .NET cryptographic library

- Facilities for interacting with Windows CryptoAPI
  - To manage CryptoAPI Key containers manually
  - To call extended functionality in CryptoAPI 2.0
- Configuration mechanism
  - The factory methods that create engine classes are driven by a configuration file that can be edited to change default algorithms and implementations
- On top of the .NET crypto API, an implementation of XML Digital Signatures is provided

69

.NET code examples

- Symmetric encryption and CryptoStreams
- Digital signatures

70

Symmetric encryption

- Creating an encrypting CryptoStream

```
SymmetricAlgorithm cipher = SymmetricAlgorithm.Create();

FileStream outStream =
  new FileStream(filename + ".enc", FileMode.Create);

CryptoStream encOutStream = new CryptoStream
  (outStream, cipher.CreateEncryptor(),
   CryptoStreamMode.Write);
```

- Now, just writing to the stream will encrypt
- Decryption is similar

Digital Signatures

- Signing:

```
AsymmetricAlgorithm cipher = DSA.Create();

AsymmetricSignatureFormatter asf =
   new DSASignatureFormatter(cipher);

SHA1 sha1 = SHA1.Create();

FileStream inStream =
   new FileStream(filename, FileMode.Open);
byte[] sig =
   asf.CreateSignature(sha1.ComputeHash(inStream));
```

Digital Signatures

- Verifying:

```
AsymmetricAlgorithm cipher = DSA.Create();
// String pubkey contains XML representation of public key
cipher.FromXmlString(pubkey);

AsymmetricSignatureDeformatter asd =
   new DSASignatureDeformatter(cipher);
SHA1 sha1 = SHA1.Create();

FileStream inStream3 =
    new FileStream(filename, FileMode.Open);
byte[] hash = sha1.ComputeHash(inStream3);

if (asd.VerifySignature(hash,sig))
    Console.WriteLine("Signature OK!");
```

73

Conclusion

- Cryptographic mechanisms should be used in such away that they are easy to evolve
  - To deal with implementation errors
  - To deal with algorithms being broken
- By structuring a library around CSP's, this can be achieved
- Java and .NET both offer a CSP based library with similar functionalities

74