Software Interfaces to Cryptographic Primitives

Frank Piessens
(Frank.Piessens@cs.kuleuven.be)
Overview

• Introduction
• Cryptographic Primitives
• Cryptographic API’s
• Key Management Issues
• Conclusion
Introduction

• Security = prevention and detection of unauthorized actions on information

• Two important cases:
  – An attacker has access to the raw bits representing the information
    => need for cryptographic techniques
  – There is a software layer between the attacker and the information
    => access control techniques
Introduction

• Cryptography builds on algorithms (primitives) that guarantee specific information security related security properties
  – E.g. Hash functions, symmetric encryption, ...
  – Precisely specifying the security properties of most primitives is intricate

• To guarantee interesting, more high-level, security properties, primitives are used in cryptographic protocols
  – E.g. Secure communication, entity authentication, ...
Cryptographic Primitives

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

- NOTE: Algorithm secrecy $\leftrightarrow$ key secrecy
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
Block ciphers and stream ciphers

- Block ciphers encrypt fixed-size input blocks
  - *Padding* may be necessary.
    - E.g. PKCS#7 padding
  - 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
Encryption modes (block ciphers)

- **Electronic Codebook (ECB)**

- **Cipher Block Chaining (CBC)**
Cleartext

DES / ECB

DES / CBC
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; standards 128, 192 or 256 bits

• RC4
  – Proprietary stream cipher of RSA Labs
Public-key Cryptography

- Key generation algorithm
- Should be secure against the same attacks as symmetric encryption
- Easier key management (see later) but slower
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
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 algorithms
  – PKCS#1 v1.5
  – OAEP
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 = $\{\text{plaintext}\}K$
  – ciphertext = $\{\text{plaintext}\}PK$
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 flavours: keyed (MAC’s) and unkeyed
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
Message Authentication Codes

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

secret key

message (any length) ➔ MAC ➔ MAC value (fixed length)
Real-world Algorithms

• Unkeyed hash functions:
  – SHA-1 (Secure Hash Algorithm)
    • Designed by NSA
    • Arbitrary-length input → 160-bit output
    • Known attacks -> now considered insecure
  – MD-5 (Message Digest)
    • By Ron Rivest
    • Arbitrary-length input → 128-bit output
    • Known attacks -> now considered insecure
  – SHA-2 / 256 and SHA-2 / 512
Real-world Algorithms

• MAC’s:
  – Any symmetric encryption of any hash function
  – Using only hash functions: $\text{MAC}_k(M) = H(k, M)$, or better: H-MAC turns any unkeyed hash in a MAC
  – DES-CBC-MAC: the last block of a CBC encryption
Digital Signatures

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

• Introduction
• Cryptographic Primitives
  • Cryptographic API’s
• Key Management Issues
• Conclusion
Cryptographic API’s

• Design principles of modern API’s:
  – Cryptographic Service Providers (CSP’s) and cryptographic frameworks

• The Java Cryptography Architecture and Extensions (JCA/JCE)

• The .NET cryptographic library

• Conclusion
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
Engine classes

• Abstraction for a cryptographic service
  – 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
Bridge pattern

```java
MessageDigest

update(byte[] input): void

digest(): byte[]
...

digest_imp()

MessageDigestImpl

update_imp(byte[] input): void

digest_imp(): byte[]
...
```
Inheritance based decoupling

MessageDigest

update(byte[] input): void
digest(): byte[]
getDigestSize(): int
...

return SHA1.digestSize

Md5

update(byte[] input): void
digest(): byte[]
getDigestSize(): int
...

SHA1

update(byte[] input): void
digest(): byte[]
getDigestSize(): int
...

SHA1-Impl1

SHA1-Impl2

update(byte[] input): void
digest(): byte[]
getDigestSize(): int
...

getDigestSize() : int
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
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
Cryptographic API’s

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

javax.crypto.*

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

KeyFactory

y = ...
p = ...
q = ...
g = ...
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

getParameterSpec()
init(paramSpec)
g = …
p = …
q = …
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
Example: creating ciphers

1: getInstance("DES/CBC/PKSC5Padding", "IAIK")

2: getProvider("IAIK")

3: getProperty("Cipher.DES")

4: CipherSpi()

5: engineSetMode("CBC")

6: engineSetPadding("PKCS5Padding")

application

Cipher

IAIK : Provider

des : CipherSpi

Security
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
Cryptographic API’s

• Design principles of modern API’s:
  – Cryptographic Service Providers (CSP’s) and cryptographic frameworks
• The Java Cryptography Architecture and Extensions (JCA/JCE)
• The .NET cryptographic library
• Conclusion
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()
ICryptoTransform and CryptoStream

• ICryptoTransforms can wrap streams
  E.g. (in read mode)
Bulk data engine classes

- SymmetricAlgorithm, with algorithm classes
  - TripleDES, DES, Rijndael, ...
- HashAlgorithm, with algorithm classes
  - SHA1, MD5, ...
- KeyedHashAlgorithm, with algorithm classes
  - HMACSHA1, MACTripleDES, ...
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
Engine classes for key generation

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

• Introduction
• Cryptographic Primitives
• Cryptographic API’s
  • Key Management Issues
• Conclusion
Key Management Issues

• Generating keys
• Key length
• Storing keys
• Key establishment
• Key renewal
• Key disposal
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
  - | Key Length | Cracking Time |
    |-----------|--------------|
    | 56 bits   | 1 hour       |
    | 64 bits   | 10 days      |
    | 128 bits  | $10^{17}$ years |
- **Public-key algorithms**

<table>
<thead>
<tr>
<th>Year</th>
<th>vs. Individual</th>
<th>vs. Corporation</th>
<th>vs. Government</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1024</td>
<td>1280</td>
<td>1536</td>
</tr>
<tr>
<td>2005</td>
<td>1280</td>
<td>1536</td>
<td>2048</td>
</tr>
<tr>
<td>2010</td>
<td>1280</td>
<td>1536</td>
<td>2048</td>
</tr>
</tbody>
</table>
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
Key Distribution

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

![Diagram]

- Alice sends \( \{K\}K_A \) to KDC
- KDC sends \( \{K\}K_A \) and \( \{K\}K_B \) to Bob
- Bob sends \( \{K\}K_B \) to Alice
Key Distribution

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

  Alice

  $$\text{P}$$

  Bob

  $$\text{PK}$$

  $${\{M}\text{PK}_B}$$

  Public key?

  – Man-in-the-middle attack!
How can Alice be sure she got Bob’s public key?

- Solution: Certificates
  - Public Key Infrastructure (PKI)
- Discussed later
Key renewal

• **Best practice:**
  – Limit the amount of data encrypted with a single key
  – Limit the amount of time a key is in use

• **Hence:**
  – Need for mechanisms to renew keys
Key disposal

• Once a key is no longer used, what should happen?
  – Short-term keys:
    • Dispose in a secure way
  – Long-term keys:
    • Encryption:
      – Reencrypt old data, or store key securely
    • Signing
      – Signing key should be disposed of securely
      – Verification key should be stored securely
Conclusion

• Good key management is essential to achieve any security from cryptography

• Inappropriate
  – Key generation
  – Key storage
  – Or key establishment

is often the cause of security breaches
Overview

• Introduction
• Cryptographic Primitives
• Cryptographic API’s
• Key Management Issues
• Conclusion
Conclusion

• Cryptographic primitives offer well-defined but complex security guarantees
  – Precisely saying what security a crypto primitive offers is non-trivial

• As a consequence, cryptographic primitives are hard to use correctly
  – Mainstream developers should typically **not** use them
  – Use API to higher-level protocols instead