Software Interfaces to Cryptographic Algorithms

Pieter Philippaerts
Pieter.Philippaerts@cs.kuleuven.be

Based on slides by Frank Piessens
Overview

- Design Principles
- The native Windows CryptoAPI
  - Cryptography API: Next Generation (CNG)
- The Java Cryptography Architecture and Extensions (JCA/JCE)
- The .NET Cryptographic Library
- The OpenBSD Cryptographic Framework
- Key management issues
- Conclusion
Crypto Timeline

- 1998: DES brute-forced in 22 hours
- 2000: AES selected as NIST standard
- 2001: RC4 weakness breaks WEP
- 2002: end-of-life of DES (inc. TripleDES), SHA2 introduced
- 2004: MD5, RIPEMD broken
- 2005: SHA1 broken
- 2012: SHA3 introduced
Design Requirements

- **New algorithms get introduced**
  - The architecture should be extensible

- **Algorithms get broken**
  - Developers should be able to easily replace one algorithm with another
Design Principles

- **Algorithm independence**
  - Engine classes / Factory methods

- **Implementation independence**
  - Provider-based architecture

- **Implementation interoperability**
  - Transparent and opaque data types

**Bottom Line:** security mechanisms should be easy to change over time
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
Overview

- Design Principles
- The native Windows CryptoAPI
  - Cryptography API: Next Generation (CNG)
- The Java Cryptography Architecture and Extensions (JCA/JCE)
- The .NET Cryptographic Library
- The OpenBSD Cryptographic Framework
- Key management issues
- Conclusion
The Windows CryptoAPI

- Introduced with Windows 95/Windows NT4
- C-based library
- Still used by most Windows programs today
Cryptographic Service Providers

- Pluggable libraries
- Implement different cryptographic algorithms
- Own a key database

- Windows and IE ship with a number of CSPs
  - Depending from version to version and language to language
Key databases

- Stores persistent keys
- Contains a number of *key containers*
  - Has a unique name
  - One for each user
  - Applications can create new containers
- Saved in a secure file
  - With access control
  - Optional ‘*strong protection*’
Keys

- **Session keys**
  - Used for symmetric encryption
  - Volatile

- **Public/private key pairs**
  - Typically two pairs per user (one for key exchange, one for digital signatures)

- **They are opaque**
  - All you get is an identification number (handle)
  - You can export them, though
The Windows CryptoAPI

Example: encrypt data

1: csp = CryptAcquireContext (<CSP provider name>, ...)

3: key = CryptGenKey (csp, <algorithm ID>)

5: CryptSetKeyParam (key, <IV, mode, padding, ...>)

7: CryptEncrypt (key, <data>)

2: csp = CPAcquireContext (...)

4: key = CryptGenKey (csp, <algorithm ID>)

6: CryptSetKeyParam (key, <IV, mode, padding, ...>)

8: CryptEncrypt (key, <data>)
Additional support for...

- Cryptographic Message Syntax (CMS)
- Public key infrastructure
- Smart cards
- Authenticode
- XML signatures
  - Windows 7 and higher
Summary

- To add an algorithm, a new CSP must be implemented
  - Not easy

- Impossible to write algorithm independent code
  - There is no notion of ‘a default algorithm’
  - However, there are defaults for implementations

- A CSP is an island; you cannot modify its behavior
Cryptography: Next Generation

- Introduced with Windows Vista
- Aims to replace Windows CryptoAPI
  - Hence, also a C-based library
- Has benefits over the CryptoAPI
  - Easy plug-in creation, better extensibility
  - Crypto isolation
  - Support for algorithmic independence
    - In CMS, SSL/TLS, …, your application
Overview

- Design Principles
- The native Windows CryptoAPI
  - Cryptography API: Next Generation (CNG)
- The Java Cryptography Architecture and Extensions (JCA/JCE)
- The .NET Cryptographic Library
- The OpenBSD Cryptographic Framework
- Key management issues
- Conclusion
The JCA/JCE

- Java Crypto API structured as a cryptographic framework with CSPs

- Split in:
  - The Java Cryptography Architecture (JCA)
  - The Java Cryptography Extensions (JCE)

- This split is because of US export-control regulations for cryptography
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

MessageDigest

```
update(byte[] input): void
digest(): byte[]
...```

MessageDigestImpl

```
update_imp(byte[] input): void
digest_imp(): byte[]
...```

digest_imp()
Inheritance-based decoupling

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

```
Md5
update(byte[] input): void
digest(): byte[]
getDigestSize(): int
...
```

```
SHA1
update(byte[] input): void
digest(): byte[]
getDigestSize(): int
...
```

```
SHA1-Impl1
update(byte[] input): void
digest(): byte[]
getDigestSize(): int
...
```

```
SHA1-Impl2
update(byte[] input): void
digest(): byte[]
getDigestSize(): int
...
```

return SHA1.digestSize
Engine classes (JCA)

java.security.*

- MessageDigest
  hash functions
- Signature
- SecureRandom
- KeyPairGenerator
  generate new key pairs
- KeyFactory
  convert existing keys
- CertificateFactory
  generate certificates from encoded form
- KeyStore
  database of keys
- AlgorithmParameters
- AlgorithmParameterGenerator
Engine classes (JCE)

- **Cipher**
  - encryption, decryption

- **Mac**

- **KeyGenerator**
  - generate new symmetric keys

- **SecretKeyFactory**
  - convert existing keys

- **KeyAgreement**

```java
javax.crypto.*
```
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`

```java
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

```
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")

application:

Cipher

2: getProvider("IAIK")

3: getProperty("Cipher.DES")

4: CipherSpi()

5: engineSetMode("CBC")

6: engineSetPadding("PKCS5Padding")

IAIK : Provider

des : CipherSpi
Additional support

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

- **Very easy integration of new classes**
  - Inherit from the correct class

- **Cryptographic configuration**
  - To set the defaults
Overview

- Design Principles
- The native Windows CryptoAPI
  - Cryptography API: Next Generation (CNG)
- The Java Cryptography Architecture and Extensions (JCA/JCE)
- The .NET Cryptographic Library
- The OpenBSD Cryptographic Framework
- Key management issues
- 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()

Input block → ICryptoTransform → Output block
ICryptoTransform and CryptoStream

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

![Diagram showing ICryptoTransform wrapping a stream]

Resulting stream

Wrapped stream

ICryptoTransform

[Diagram details]
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, (EC)DSA and ECDH 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...

- Facilities for interacting with Windows CryptoAPI / CNG
  - To manage CryptoAPI Key containers manually
  - To call extended functionality in CryptoAPI

- 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
Differences between Java and .NET

- .NET class structure is much simpler
  - Hardly support for opaque keys
  - Wrappers around the CryptoAPI
  - Perhaps too simple?
Problems with transparent keys

SymmetricAlgorithm
- Key : byte[]
- IV : byte[]

+ Create()
+ CreateEncryptor()
+ CreateDecryptor()

Rijndael

RijndaelManaged

DES

DESCryptoServiceProvider
Problems with transparent keys

SymmetricAlgorithm algo
  = SymmetricAlgorithm.Create();

algo.Key = …

algo.IV = …

encryptor = algo.CreateEncryptor();

encryptor.TransformBlock(…);
Problems with transparent keys

- Solution: add opaque key support

<table>
<thead>
<tr>
<th>SymmetricAlgorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Key : byte[]</td>
</tr>
<tr>
<td>- IV : byte[]</td>
</tr>
<tr>
<td>+ Create()</td>
</tr>
<tr>
<td>+ CreateEncryptor()</td>
</tr>
<tr>
<td>+ CreateDecryptor()</td>
</tr>
<tr>
<td>+ FromXmlIString()</td>
</tr>
<tr>
<td>+ ToXmlIString()</td>
</tr>
</tbody>
</table>
Problems with transparent keys

SymmetricAlgorithm algo = SymmetricAlgorithm.Create();
algo.FromXmlString(…);
encryptor = algo.CreateEncryptor();
encryptor.TransformBlock(…);
Problems with wrapper code

AsymmetricAlgorithm

+ Create()
+ FromXmlString()
+ ToXmlString()
Problems with wrapper code

**RSA**

- `+Create()`
- `+EncryptValue()`
- `+DecryptValue()`
- `+ImportParameters()`
- `+ExportParameters()`

Diagram:

- **AsymmetricAlgorithm**
- **RSA**
- **RSACryptoServiceProvider**
Problems with wrapper code

RSACryptoServiceProvider

+Encrypt()
+Decrypt()
+SignData()
+VerifyData()
Problems with wrapper code

- RSAPKCS1KeyExchangeFormatter
  - +SetKey()
  - +CreateKeyExchange()

- RSAPKCS1KeyExchangeDeformatter
  - +SetKey()
  - +DecryptKeyExchange()

- AsymmetricKeyExchangeFormatter
  - RSAPKCS1KeyExchangeFormatter

- RSASOAEPKeyExchangeFormatter/Deformatter

- RSAPKCS1SignatureFormatter/Deformatter

- DSASignatureFormatter/Deformatter
Problems with wrapper code

```csharp
public byte[] CreateKeyExchange(...) {

if (rsaKey is RSACryptoServiceProvider) {
    return ((RSACryptoServiceProvider)rsaKey).Encrypt(...);
} else {
    <perform padding>
    return rsaKey.EncryptValue(<padded bytes>);
}
```

Problems with wrapper code

- Problem: only RSACryptoServiceProvider gets a ‘special’ treatment
  - Custom RSA implementations **must** support raw RSA
Problems with wrapper code

Solution:

<table>
<thead>
<tr>
<th>RSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Create()</td>
</tr>
<tr>
<td>+EncryptValue()</td>
</tr>
<tr>
<td>+DecryptValue()</td>
</tr>
<tr>
<td>+ImportParameters()</td>
</tr>
<tr>
<td>+ExportParameters()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RSACryptoServiceProvider</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Encrypt()</td>
</tr>
<tr>
<td>+Decrypt()</td>
</tr>
<tr>
<td>+SignData()</td>
</tr>
<tr>
<td>+VerifyData()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Create()</td>
</tr>
<tr>
<td>+EncryptValue()</td>
</tr>
<tr>
<td>+DecryptValue()</td>
</tr>
<tr>
<td>+ImportParameters()</td>
</tr>
<tr>
<td>+ExportParameters()</td>
</tr>
<tr>
<td>+Encrypt()</td>
</tr>
<tr>
<td>+Decrypt()</td>
</tr>
<tr>
<td>+SignData()</td>
</tr>
<tr>
<td>+VerifyData()</td>
</tr>
<tr>
<td>+SupportsRaw()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RSACryptoServiceProvider</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Problems with wrapper code

```java
public byte[] CreateKeyExchange(...) {

    if (!rsaKey.SupportsRaw) {
        return rsaKey.Encrypt(...);
    } else {
        <perform padding>
        return rsaKey.EncryptValue(<padded bytes>);
    }

    // subscript
}
```
Summary

- Extensible class hierarchy
- Cryptographic configuration support
- Some small issues
  - Can be resolved with some minor tweaks
Overview

- **Design Principles**
- **The native Windows CryptoAPI**
  - Cryptography API: Next Generation (CNG)
- **The Java Cryptography Architecture and Extensions (JCA/JCE)**
- **The .NET Cryptographic Library**
- **The OpenBSD Cryptographic Framework**
- **Key management issues**
- **Conclusion**
OpenBSD

- NetBSD spin-off
- Focuses on security
  - Cryptography is the cornerstone of the system
  - Defensive programming
    - Periodically go through source
The OpenBSD Crypto Framework

- OpenBSD Cryptographic Framework (OCF)
  - “Asynchronous service virtualization layer”
  - Resides in kernel
  - Offers uniform access to crypto hardware
  - Used by
    - Producers (crypto hardware)
    - Consumers (other kernel modules)
The OpenBSD Crypto Framework

- **Two modi operandi**
  - Session-based
    - Symmetric crypto, hashing
    - Session caching features
  - Individual operations
    - Asymmetric crypto
The OpenBSD Crypto Framework

- **Producers**
  - Are drivers
  - Registers with OCF
    - Supported algorithms
    - Other capabilities (chaining, RNG, …)
  - One pseudo-driver
    - Software crypto
The OpenBSD Crypto Framework

- Consumers
  - Other modules in kernel (e.g. IPSec)
  - Send asynchronous requests to the OCF
    - Get notified when the work is complete
    - Synchronous requests not supported
The OpenBSD Crypto Framework

- A consumer doesn’t know which producer it’s talking to
  - The OCF takes care of this automatically
  - Enables load-balancing
  - Enables session-migration
    - When hardware is added/removed (i.e. PCMCIA card)
    - On-demand
  - Important difference between OCF and Java/.NET/CryptoAPI
The OpenBSD Crypto Framework

- This is a kernel framework
- User-level support is added through the /dev/crypto interface
  - Synchronous!
  - Based on ioctl() calls
  - Not very user friendly
  - Frameworks like OpenSSL offer abstractions over /dev/crypto
Summary

- Extensible (through device drivers)
- Crypto configuration is done by the framework behind the scenes
  - Applications do not see the different ‘CSPs’
Overview

- Design Principles
  - The native Windows CryptoAPI
    - Cryptography API: Next Generation (CNG)
  - The Java Cryptography Architecture and Extensions (JCA/JCE)
  - The .NET Cryptographic Library
  - The OpenBSD Cryptographic Framework
- 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 $\rightarrow$ dictionary attack!
  - Better: hash of entire pass-phrase
  - Machine-generated $\rightarrow$ use cryptographically secure pseudo-random generator
Key length

- **Trade-off:** information value ↔ cracking cost

- **Symmetric algorithms**
  - $1,000,000$ investment in VLSI-implementation

<table>
<thead>
<tr>
<th>Key length</th>
<th>56 bits</th>
<th>64 bits</th>
<th>128 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 hour</td>
<td>10 days</td>
<td>$10^{17}$ years</td>
</tr>
</tbody>
</table>

- **RSA**

<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):

```
Alice  -> KDC
       | {K}K_A, {K}K_B
       v
Bob?  <- KDC
       | Ka, Kb
       v
{K}K_B  -> Bob
```
Key distribution

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

\[
\text{Public key?} \quad PK_B \quad \text{？}
\begin{align*}
\text{Bob} & \quad \{M\}PK_B \\
\text{Alice} & \quad PK_A
\end{align*}
\]

– 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:
    - Re-encrypt old data, or store key securely
  - Signing:
    - Signing key should be disposed of securely
    - Verification key should be stored securely
Summary

- 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

- Design Principles
- The native Windows CryptoAPI
  - Cryptography API: Next Generation (CNG)
- The Java Cryptography Architecture and Extensions (JCA/JCE)
- The .NET Cryptographic Library
- The OpenBSD Cryptographic Framework
- 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