Ethical Hacking

Die Netz Security

Crypto Cheat Sheet

Best Practices: Cryptography

Manu Carus

http://www.ethical-hacking.de/
mailto:manu.carus@ethical-hacking.de

Ethical Hacking

Die Netz Security

Table of Content

1 CRYPTOGRAPHY 5

1.1 OBJECTIVES OF CRYPTOGRAPHY 5
1.2 BEST PRACTICES 7
1.2.1 CRYPTOGRAPHIC AGILITY 7
1.3 WORST PRACTICES 8
1.3.1 SECURITY BY OBSCURITY 8
1.3.2 AD HOC ALGORITHMS 8

2 RANDOM NUMBERS 9

2.1 SUPPORT IN LIBRARIES AND FRAMEWORKS 9
2.1.1 OPENSSL 9
2.1.2 JAVA CRYPTOGRAPHY ARCHITECTURE 9
2.1.3 .NET 9
2.1.4 CRYPTOAPI/CAPICOM 9
2.2 BEST PRACTICES 10
2.2.1 CRYPTOGRAPHICALLY SECURE GUIDS 10
2.3 RULES OF THUMB 10
2.3.1 EFFORT INVOLVED IN COMPUTING CRYPTOGRAPHICALLY SECURE RANDOM NUMBERS 10

3 HASHING ALGORITHMS 11

3.1 REQUIREMENTS PLACED ON A SECURE HASHING ALGORITHM 12
3.2 APPLICATION AREAS 12
3.2.1 CHECKSUMS 12
3.2.2 DIGITAL SIGNATURES 12
3.2.3 COMPARISON OF LARGE INPUTS 12
3.3 SECURE HASHING METHODS 13
3.4 SUPPORT IN LIBRARIES AND FRAMEWORKS 13
3.4.1 OPENSSL 13
3.4.2 JAVA CRYPTOGRAPHY ARCHITECTURE 13
3.4.3 .NET 13
3.4.4 CRYPTOAPI/CAPICOM 13
3.5 BEST PRACTICES 14
3.5.1 SHA-256 14
3.5.2 HASH VALUES THAT ARE 256 BITS LONG OR LONGER 14
3.5.3 HASHING RATHER THAN ENCRYPTION 14
3.5.4 SALTED HASHING RATHER THAN HASHING FOR SMALL INPUT 14
3.5.5 COMPARISON OF LONG INPUTS 14
3.5.6 INTEGRITY CHECK 15
3.5.7 TRANSFERRING HASH VALUES IN ENCRYPTED FORM 15
3.5.8 DIGITAL SIGNATURE RATHER THAN HASHING 15
3.6 WORST PRACTICES 15
3.6.1 MD5 15
3.6.2 SHA-1 15
3.7 RULES OF THUMB 16
3.7.1 COLLISION-FREENESS 16

4 SALTED HASHING 17

5 MESSAGE AUTHENTICATION CODES 19
5.1 SECURE MAC METHODS 19
5.2 SUPPORT IN LIBRARIES AND FRAMEWORKS 20
5.2.1 OPENSSL 20
5.2.2 JAVA CRYPTOGRAPHY ARCHITECTURE 20
5.2.3 .NET 20
5.2.4 CRYPTOAPI/CAPICOM 20
5.3 BEST PRACTICES 21
5.3.1 HMACSHA256 21
5.3.2 USE OF LONG KEYS 21
5.3.3 DIGITAL SIGNATURES 21

6 SYMMETRIC ENCRYPTION 22
6.1 KEY EXCHANGE 23
6.2 SECURE SYMMETRIC ALGORITHMS 23
6.3 SUPPORT IN LIBRARIES AND FRAMEWORKS 24
6.3.1 OPENSSL 24
6.3.2 JAVA CRYPTOGRAPHY ARCHITECTURE 24
6.3.3 .NET 24
6.3.4 CRYPTOAPI/CAPICOM 25
6.4 BEST PRACTICES 25
6.4.1 AES-256 25
6.4.2 KEYS WHICH ARE 128 BITS LONG AND LONGER 25
6.4.3 KEYS WHICH ARE 256 BITS LONG 25
6.4.4 SSL RATHER THAN SYMMETRIC ENCRYPTION 25
6.4.5 HYBRID ENCRYPTION RATHER THAN SYMMETRIC ENCRYPTION 25
6.4.6 KEY GENERATION 25
6.4.7 INITIALIZATION VECTOR 26
6.4.8 CIPHER BLOCK CHAINING (CBC) 26
6.4.9 PKCS #7 PADDING 26
6.4.10 DERIVING A KEY FROM A PASSWORD 27
6.4.11 STRONG PASSWORDS 27
6.5 WORST PRACTICES 28
6.5.1 DES 28
6.5.2 ELECTRONIC CODE BOOK (ECB) 28
6.5.3 ENCRYPTING DATA TWICE ON A COMMUNICATION CHANNEL WHICH HAS END-TO-END PROTECTION 28
6.5.4 COMPRESSING ENCRYPTED DATA 28
6.6 RULES OF THUMB 29
6.6.1 AES ENCRYPTS MUCH FASTER THAN TRIPLEDES 29
6.6.2 TWOFISH RATHER THAN BLOWFISH 29
6.6.3 KEY STRENGTH OF TRIPLE DES 29
6.6.4 SYNCHRONIZATION OF ENCODING 29

7 ASYMMETRIC ENCRYPTION 30
7.1 SECRECY 33
7.2 SECURE ASYMMETRIC ENCRYPTION ALGORITHMS 34
7.3 SUPPORT IN LIBRARIES AND FRAMEWORKS 34
7.3.1 OPENSSL 34
7.3.2 JAVA CRYPTOGRAPHY ARCHITECTURE 34
7.3.3 .NET 34

7.3.4 CryptoAPI/CAPICOM 34
7.4 Best practices 35
7.4.1 RSA-2048 35
7.4.2 Hybrid encryption 35
7.4.3 Sending a message to several recipients 35
7.5 Worst practices 36
7.5.1 RSA-1024 36
7.5.2 Encrypting contents asymmetrically 36

8 Hybrid encryption 37

8.1 Achieving cryptographical objectives 39
***8.2* Support in libraries and frameworks 39**
8.2.1 OpenSSL 39
8.3 Best practices 39
8.3.1 Fast and secure 39

9 Digital signature 40

9.1 Secure signature algorithms 41
***9.2* Support in libraries and frameworks 42**
9.2.1 OpenSSL 42
9.2.2 Java Cryptography Architecture 42
9.2.3 .NET 42
9.2.4 CryptoAPI/CAPICOM 43
9.3 Best practices 43
9.3.1 DSA rather than RSA 43
9.3.2 Signature algorithms using hash values which are 160 bits long or more 43
9.3.3 Keeping a record of all signed documents 43

10 Key exchange 44

***10.1* Support in libraries and frameworks 45**
10.1.1 OpenSSL 45
10.1.2 Java Cryptography Architecture 45
10.1.3 .NET 45
10.1.4 CryptoAPI/CAPICOM 45
10.2 Best practices 45
10.2.1 Diffie-Hellman only in conjunction with mutual authentication 45
10.3 Rules of thumb 46
10.3.1 Effort involved in computing Diffie-Hellman parameters 46

11 In-memory encryption 46

12 Severability clauses 47

12.1 Cryptographically weak algorithms 47
12.2 Commandment of caution 47

13 List of abbreviations 48

1 Cryptography

Definitions

Cryptography is the science of hiding information.

The term is contrasted with *cryptanalysis* where the objective is to analyze and break encryption methods which have hitherto been considered secure.

Both these sciences are branches of *cryptology*.

1.1 Objectives of cryptography

Modern cryptography defines four main objectives in order to protect confidential information:

Secrecy

"keeping information secret"

Sensitive data must be concealed from others. Only explicitly authorized persons must be able to read confidential data or obtain knowledge about its content.

Integrity

"knowing that information hasn't been tampered with"

The recipient of a message must be able to ascertain whether the message was modified after it was generated and before it was received.

Authentication

"knowing the origin and destination of information"

Both the originator and recipient of a message must be unambiguously identifiable.

Non-repudiation

"knowing that information, once sent, cannot be retracted or denied".

The originator of a message must not be able to repudiate his authorship. It must be possible to substantiate authorship to third parties.

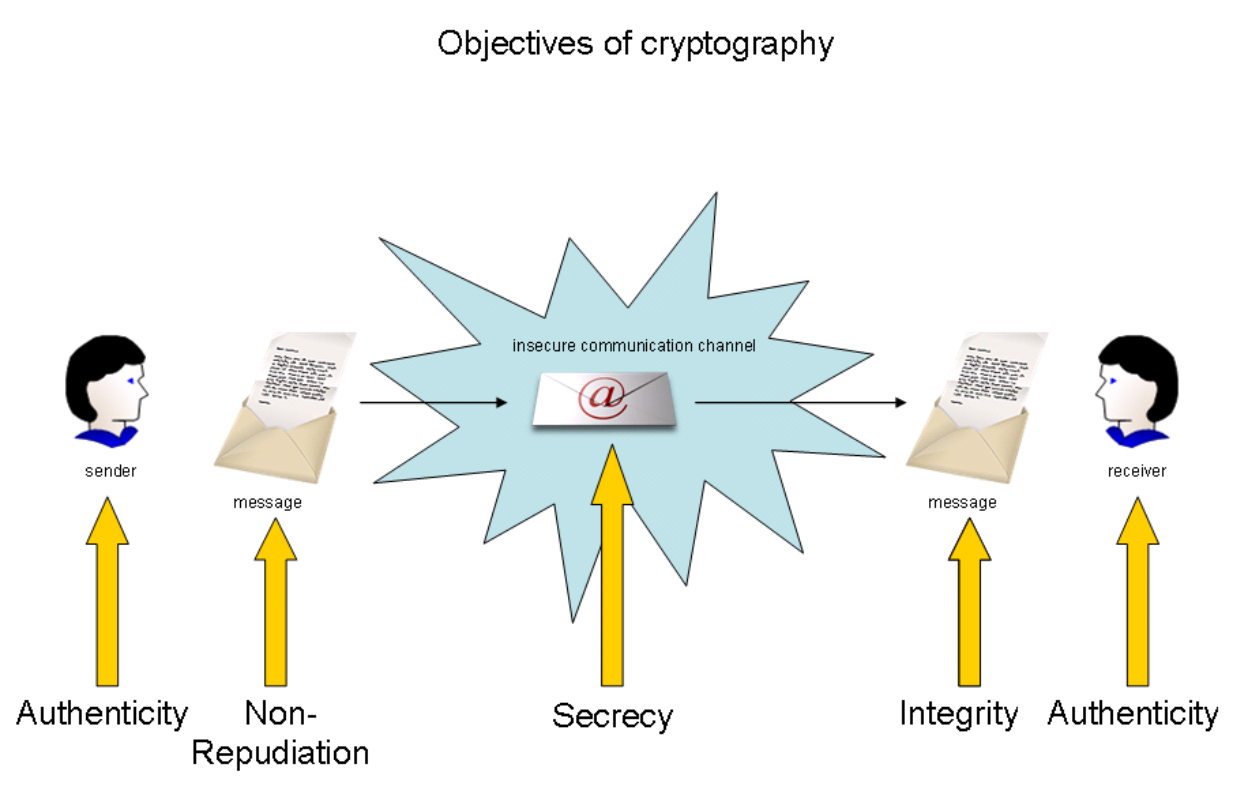

Fig.: Objectives of cryptography

Requirement

A "secure" application must achieve these four objectives of cryptography!

These objectives can be achieved by using the methods that are presented below. Nevertheless, on their own, they do not make it possible to guarantee complete protection of confidential data. In fact, each of the technologies used opens up the persons involved and systems as well as the processes applied to further attack vectors. For example: Man-in-the-middle attack, replay attacks, injection flaws.

1.2 Best practices

1.2.1 Cryptographic agility

Because cryptanalysts are constantly probing both known and new cryptographic algorithms for potential weaknesses, situations can occur where a cryptographic algorithm which has hitherto been classified as secure has to be replaced as quickly as possible. This was the case, for instance, in December 2008 when it was proved that the MD5 hash algorithm had a collision weakness.

In such a case, an application must be able to switch to a stronger cryptographic algorithm quickly and flexibly, if possible without having to compile the underlying source code and ship it again.

Example (C#)

```
string cipherName = ConfigurationManager.AppSettings["Cipher"]; // AES-256
SymmetricAlgorithm cipher = SymmetricAlgorithm.Create(cipherName);
```

instead of

```
DESCryptoServiceProvider cipher = new DESCryptoServiceProvider();
```

The cryptographic algorithms that are to be used can, for instance, be named in a configuration file. The latter must be protected against unauthorized modifications, for example by a digital signature or by verifying a HMAC because, otherwise, an attacker could swap a secure cryptographic algorithm for a weak one (e.g., AES-256 ⇨ DES).

1.3 Worst practices

1.3.1 Security by obscurity

Secrets must not be "cloaked"; they must remain secret. This is why passwords, keys or other cryptographically sensitive information must not be "hidden" in source code or in other essentially accessible locations.

It must be assumed that supposedly hidden hiding places will always be found, especially by third parties who are not meant to know about such hiding places.

Example

Binary code can be decompiled; an attacker can not only find "secret" strings in this way, they can also analyze the entire processing logic of an application. This is why secrets must never be hard-coded. Concealment tactics such as storing byte arrays instead of character strings in the source code do not make it harder to read "secrets".

Example

"Secret" debug switches in a release version get around; if a help desk employee has to tell a call center agent how to set a "secret" switch when dealing with an incident by phone just once, the "secret" is out!

Not only that, it can be assumed that an attacker has extensive knowledge of the cryptographic algorithms used and their key sizes and block sizes as well as knowledge about techniques for generating random numbers and tickets as well as keys which are generated on the basis of random numbers.

1.3.2 Ad hoc algorithms

Improvised algorithms devised by a programmer must not be used for encryption, hashing and generating random numbers and the like: They have not undergone cryptographic analysis by highly trained cryptanalysts! Issues such as key sizes, chaining modes and internal processing of data by the algorithm demand thorough analysis to point out possible weaknesses. Tried-and-tested algorithms such as AES use several rounds of expansion, permutation, relocation, folding and mirroring, and the number of rounds depends on the key size and block size.

2 Random numbers

A finite algorithm can be represented as a finite automaton. Identical inputs generate identical outputs. Conventionally constructed "random numbers" are therefore based on deterministic parameters such as system time, MAC address, CPU instruction pointer, etc. They are therefore not really random but predictable to a certain extent.

In contrast, cryptographic algorithms require genuine random numbers: Keys, initialization vectors and salt values must not be predictable under any circumstances. Special algorithms for generating cryptographically secure random numbers must therefore be used in the context of encryption.

2.1 Support in libraries and frameworks

2.1.1 OpenSSL

```
openssl rand
```

2.1.2 Java Cryptography Architecture

```
java.security.SecureRandom    (algorithm ∈ {SHA1PRNG})
```

2.1.3 .NET

```
System.Security.Cryptography.RNGCryptoServiceProvider
```

2.1.4 CryptoAPI/CAPICOM

```
CryptGenKey()
CryptGenRandom()
```

2.2 Best practices

2.2.1 Cryptographically secure GUIDs

Globally Unique Identifiers (GUIDs) are globally unique 128-bit numbers (16 bytes) which are often used as session IDs or the like.

It is not known whether a particular GUID is cryptographically secure! The standard algorithms for generating a GUID must therefore not be used in a cryptographical context:

C++: ~~`GUID`~~

.NET: ~~`System.Guid`~~

Instead, a byte array having a length of 16 with cryptographically secure random numbers must be generated and converted to the format of a GUID.

Example (C++)

```
...
array<Byte>^ randomNumber = gcnew array<Byte>(16);
RNGCryptoServiceProvider^ cryptoProvider = gcnew RNGCryptoServiceProvider();
cryptoProvider->GetBytes(randomNumber);
...
```

Example (C#)

```
...
byte[] randomBytes = new byte[16];
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
rng.GetBytes(randomBytes);
Guid guid = new Guid(randomBytes);
...
```

Example (Java)

The Java class `java.util.UUID` which already generates cryptographically secure random numbers is an exception:

```
...
UUID uuid = java.util.UUID.randomUUID();
...
```

2.3 Rules of thumb

2.3.1 Effort involved in computing cryptographically secure random numbers

Computing cryptographically secure random numbers requires approximately 10 times more effort than computing classic random numbers.

3 Hashing algorithms

In English, "to hash" means "to chop up".

A hashing algorithm h is a function which represents a character string z* of any length as a comparatively small, fixed-length numeric value w which is as unique as possible.

$h: Z^* \rightarrow \{0,1\}^k$ where Z := { z | z ∈ Unicode } and k>0 fixed

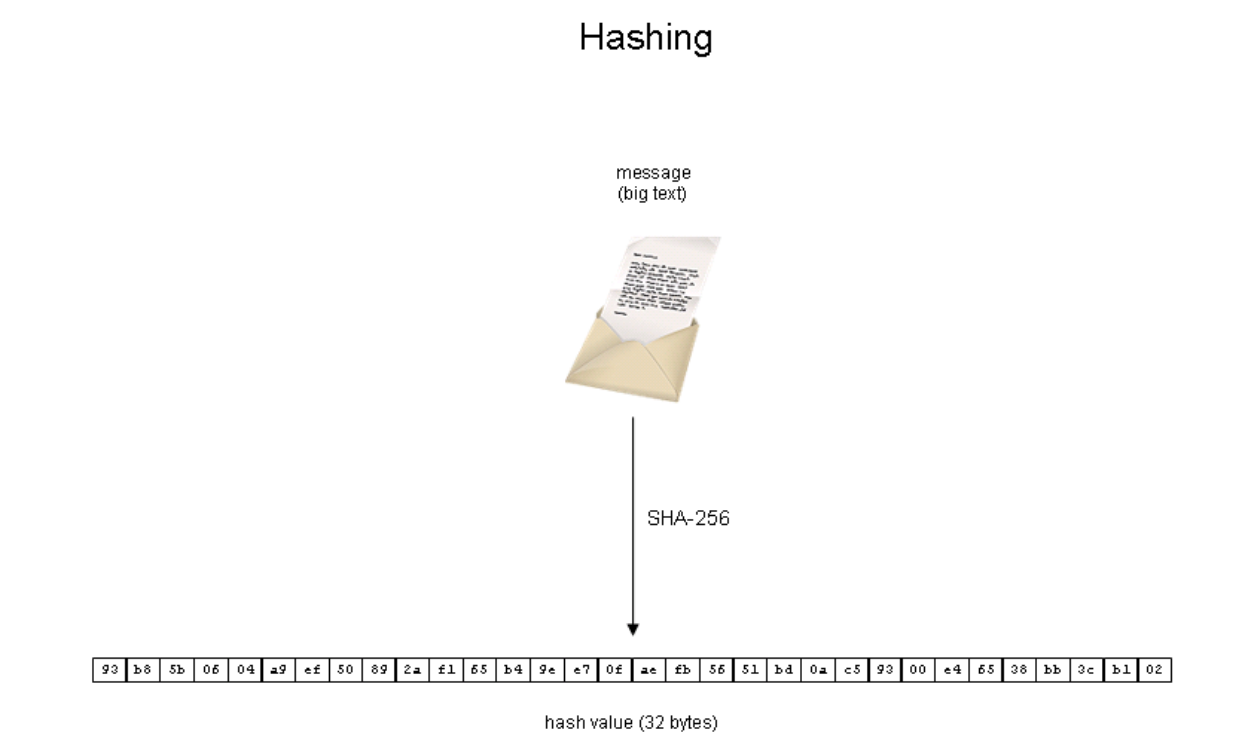

Fig.: Hashing

A hash value represents its input in the form of an easy-to-handle, short numeric value and is therefore also referred to as a "fingerprint".

3.1 Requirements placed on a secure hashing algorithm

Cryptographically secure hashing algorithms must meet the following requirements:

- For a given hash value h(z*) = w, it must be practically impossible to find the input value z* (one-way function).

- There must be practically no collisions, i.e., $z^* \neq z'^* \Rightarrow h(z^*) \neq h(z'^*)$.

- Avalanche effect: The hashing function must react as sensitively as possible to even a small change in input (small change, big effect). For instance, inserting a single character in the input must produce a more-than-30% change in the bits of the resulting hash value.

3.2 Application areas

3.2.1 Checksums

If the recipient calculates the hash value of a file and compares this checksum to the hash value published by the sender, this makes it possible to ensure the integrity of the file.

3.2.2 Digital signatures

A digital signature is the encrypted hash value of information encrypted using the information owner's private key.

3.2.3 Comparison of large inputs

Rather than comparing two large inputs one byte at a time, it can be more efficient to compare the hash values of the two inputs.

3.3 Secure hashing methods

Algorithm		Hash value	
SHA-256	Secure hashing algorithm	256 bit	32 bytes
SHA-384		384 bit	48 bytes
SHA-512		512 bit	64 bytes
RIPEMD-160	RACE Integrity Primitives Evaluation Message Digest	160 bit	20 bytes

Last revised: 12/2012

3.4 Support in libraries and frameworks

3.4.1 OpenSSL

```
openssl dgst -alg (alg ∈ {sha256, sha384, sha512, ripemd160})
```

3.4.2 Java Cryptography Architecture

```
java.security.MessageDigest   (algorithm ∈ {SHA-256, SHA-384, SHA-512})
```

3.4.3 .NET

```
System.Security.Cryptography.SHA256Managed
System.Security.Cryptography.SHA384Managed
System.Security.Cryptography.SHA512Managed
System.Security.Cryptography.RIPEMD160Managed
```

3.4.4 CryptoAPI/CAPICOM

```
CryptCreateHash()
CryptHashData()
CryptHashMessage()
CryptHashPublicKeyInfo()
CryptHashSessionKey()
CryptVerifyDetachedMessageHash()
CryptVerifyMessageHash()

ALG_ID ∈ {CALG_SHA_256, CALG_SHA_384, CALG_SHA-512}
```

3.5 Best practices

3.5.1 SHA-256

SHA-256 is cryptographically secure, easy to handle and is supported by all established frameworks.

3.5.2 Hash values that are 256 bits long or longer

The cryptographic strength of a hash value is comparable to half the key strength of a symmetric algorithm. For instance, relatively, SHA-256 is as resistant to brute force attacks as AES-128. Because symmetric algorithms with key sizes of less than 100 bits are known to be weak, hash values which are at least 256 bits long (32 bytes) must be used.

3.5.3 Hashing rather than encryption

Hashing algorithms are one-way functions, i.e., it is not possible to reconstruct the underlying input from a hash value. This is why encryption should be dispensed with whenever possible and the hash value of confidential information should be computed instead.

Example

Passwords are the exclusive property of their respective owner and it must not be possible to reconstruct them using a computer.

3.5.4 Salted hashing rather than hashing for small input

Short inputs such as passwords, for instance, must be protected through an additional salt value (see Section entitled "Salted hashing").

3.5.5 Comparison of long inputs

If long inputs have to be compared to each other, it is more practicable to calculate their hash values and then compare the hash values.

Example

Checking for duplicate incoming orders which are placed in a queue in the form of XML documents.

3.5.6 Integrity check

The files which comprise an application must be subjected to an integrity check before they are used. This applies to executables, libraries and configuration files. Damaging manipulation after the build process can be detected this way.

Integrity checking is performed by computing the hash value of a file before using an application and comparing it to the hash value which was previously computed for this file within the build process by the build master.

3.5.7 Transferring hash values in encrypted form

Hash values which are used to check the integrity of a file or download must be transmitted in encrypted form. Otherwise an attacker could intercept, modify, hash and resend the transmitted data.

3.5.8 Digital signature rather than hashing

Digital signatures should be used to verify code and downloads rather than checksums. Private keys do not have to be transmitted and, overall, are more effectively protected against attacks. In addition, *code signing* by X.509v3 certificates is supported.

3.6 Worst practices

3.6.1 MD5

MD5 was cryptographically cracked in December 2008: MD5 has a collision weakness which makes it possible to manipulate input in such a way that the resulting MD5 hash value equals to the MD5 hash value of different input! Although this algorithm is in widespread practical use[1], it must now no longer be used.

3.6.2 SHA-1

Although SHA-1 is in widespread practical use and supported by all established frameworks, SHA-1 must not be used due to known collision attacks and because of its excessively short hash value length (128 bits). If no better hashing method is available, the severability clause then applies (see section at the end of this document).

[1] in PGP and for calculating file checksums for integrity check purposes

3.7 Rules of thumb

3.7.1 Collision-freeness

Hash values with a length of 256 can represent up to 2^{256} different documents. This is roughly equivalent to a value range of 10^{76} different hash values.

Usually, there are no duplicates among the 10^{76} documents. As long as the selected hashing algorithm is (practically) collision free, the use of hash values can be regarded as cryptographically secure in practice for checking the integrity of messages and files.

4 Salted hashing

Hashing methods prevent the possibility to reconstruct entered data retrospectively. This is an indispensable feature for passwords which are used during a logon procedure.

Nevertheless, with considerable computing effort and a high degree of parallelism, it is possible to compute the hash values of all (!) possible inputs (*brute force attack*). So-called *rainbow tables*, lookup reference tables which reveal the associated original data when a hash value is entered, are the result of such complex computing. Rainbow tables are computed in distributed bot networks which can be freely queried on the Internet.

Because identical passwords imply identical hash values, hash values for passwords must be specially protected against this type of attack. To achieve this, salt values are "sprinkled" into the computed hash value (*salted hashing*). This causes a disproportionate increase in the computing effort needed to mount brute force attacks.

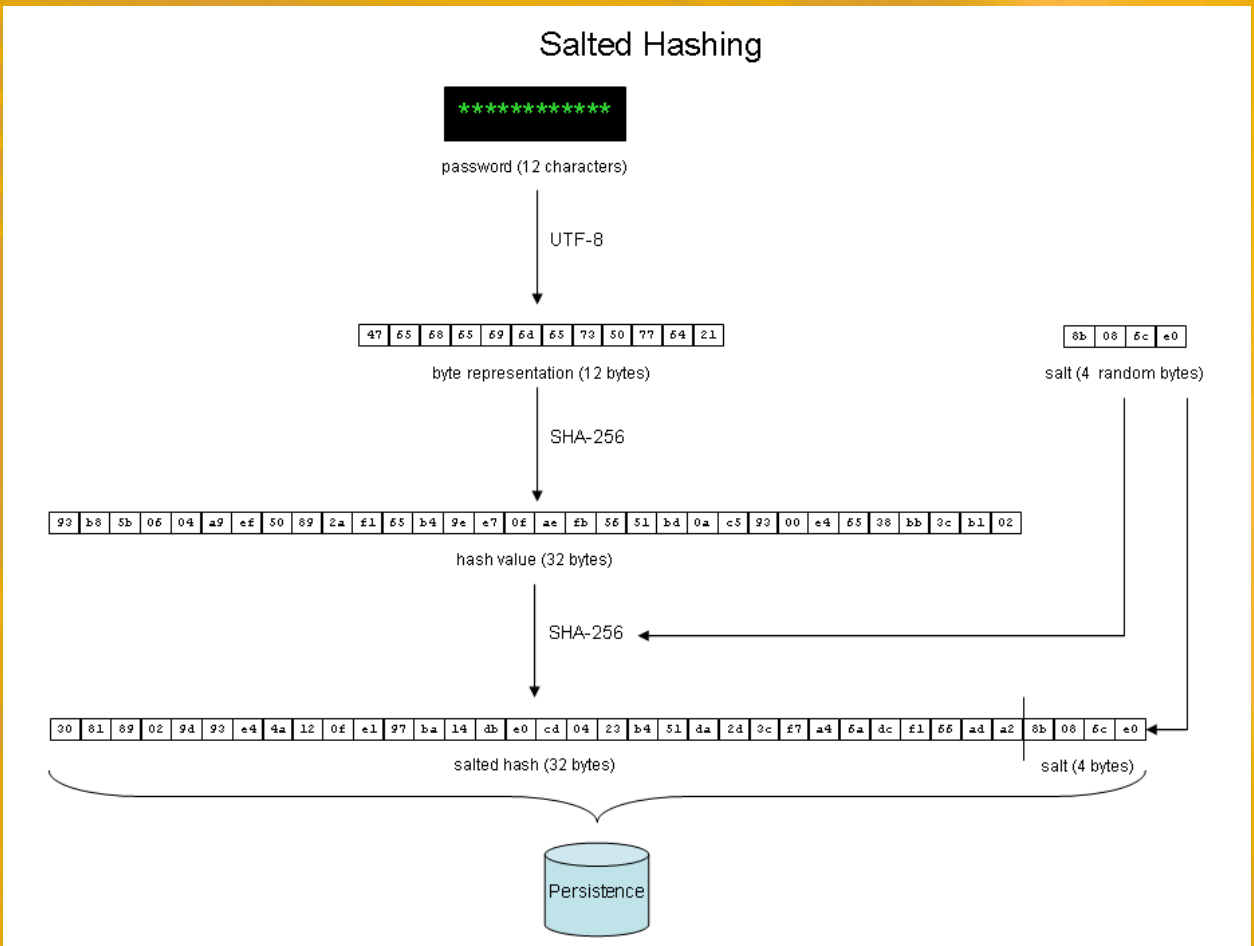

Fig.: Salted hashing

In the case of salted hashing, a password is represented as a byte sequence. Once the hash value for this byte sequence has been computed, another fixed-length byte sequence is generated and populated with random values (so-called *salt*). This salt value is fed into another hash calculation for the hash value which has already been computed. The result is referred to as a *salted hash*. Both byte sequences, the salted hash and the actual salt value, are then stored as a concatenated byte sequence.

The salt is generated individually for every password. This way, different passwords give different hash values. A brute force attack can no longer be made on the hashing algorithm in its entirety; specific rainbow tables have to be generated for each individual salted hash, taking into account the individual salt values.

Recommendations:

Salts must be at least four bytes long, eight bytes long is recommended.

Salt values should be stored separately from hash values in order to achieve a higher level of security.

5 Message Authentication Codes

An alternative approach can be adopted, instead of digital signatures, in order to protect messages against unnoticed manipulation: So-called *Message Authentication Codes*. This involves an authentication code which is computed and encrypted by the sender of a message. If the sender's decrypted authentication code matches the authentication code computed by the recipient, the message was (presumably) not tampered with en route from the sender to the recipient. The actual authentication code is protected by symmetric encryption during transmission to the recipient. The sender and recipient therefore share a secret - the symmetric key. This is why the authentication code for this method is used to authenticate both communication partners.

A Message Authentication Code (MAC) signs a message in this sense, thus ensuring its integrity and authenticity. The actual message itself is, however, not encrypted.

From a technical standpoint, a MAC is the symmetrically encrypted hash value of a message. Because the symmetric key directly comprises the hashing method, different keys result in different MACs using the same hashing method.

5.1 Secure MAC methods

Hash-based Message Authentication Code (HMAC) combines secure symmetric encryption with a secure hashing method (e.g., HMAC-SHA-256). HMAC is used in TLS and IPSec protocols among others.

In contrast to hash-based MACs, a **Cipher-based Message Authentication Code (CMAC)** is based on a symmetric block cipher method (e.g., AES-CMAC). The same security objectives as in the case of HMAC are achieved. Usually, CMACs are used precisely in situations where symmetric encryption methods are more efficiently available than hashing methods (e.g., Crypto Accelerator).

MAC-3DES-CBC uses the TripleDES method with cipher block chaining for symmetric encryption.

Both methods are also called *Keyed hash algorithms* because hash computing is based on a (symmetric) key.

5.2 Support in libraries and frameworks

5.2.1 OpenSSL

```
openssl dgst -alg -hmac        (alg ∈ {sha256, sha384, sha512, ripemd160})
```

5.2.2 Java Cryptography Architecture

```
javax.crypto.KeyGenerator   (algorithm ∈ {HmacSHA256, HmacSHA384,
                                          HmacSHA512})
javax.crypto.Mac            (algorithm ∈ {HmacSHA256, HmacSHA384,
                                          HmacSHA512})
```

5.2.3 .NET

```
System.Security.Cryptography.KeyedHashAlgorithm.HMACSHA256
System.Security.Cryptography.KeyedHashAlgorithm.HMACSHA384
System.Security.Cryptography.KeyedHashAlgorithm.HMACSHA512
System.Security.Cryptography.KeyedHashAlgorithm.HMACRIPEMD160
System.Security.Cryptography.KeyedHashAlgorithm.MACTripleDES
```

5.2.4 CryptoAPI/CAPICOM

```
CryptCreateHash()
CryptHashData()
CryptHashMessage()
CryptHashPublicKeyInfo()
CryptHashSessionKey()
CryptVerifyDetachedMessageHash()
CryptVerifyMessageHash()

ALG_ID ∈ {CALG_HMAC, CALG_MAC}
```

5.3 Best practices

5.3.1 HMACSHA256

HMACSHA256 is cryptographically secure, easy to handle and is supported by all established frameworks.

Some Hardware Security Modules (HSM) also support the AES-CMAC-256 algorithm which is also classified as secure.

5.3.2 Use of long keys

Long sequences of cryptographically secure random numbers (e.g., 24 bytes) must be used as keys.

If a password is used as a template for the key, a password security policy which stipulates strong passwords must first be followed. Strong passwords are at least 12 characters long, comprise upper and lower case letters and special characters, cannot be guessed and are not listed in any dictionary. In addition, the *PBKDF2* algorithm must be used to derive a key from the specified password (see section entitled Symmetric encryption, Best practices). Improvised own coding is not permitted.

5.3.3 Digital signatures

Scenarios which demand high levels of security must be protected using digital signatures, not a MAC method. Digital signatures do not require a common secret shared by the sender and recipient of a message; they use distributed keys.

6 Symmetric encryption

With symmetric encryption, the sender and recipient of a message use the same key.

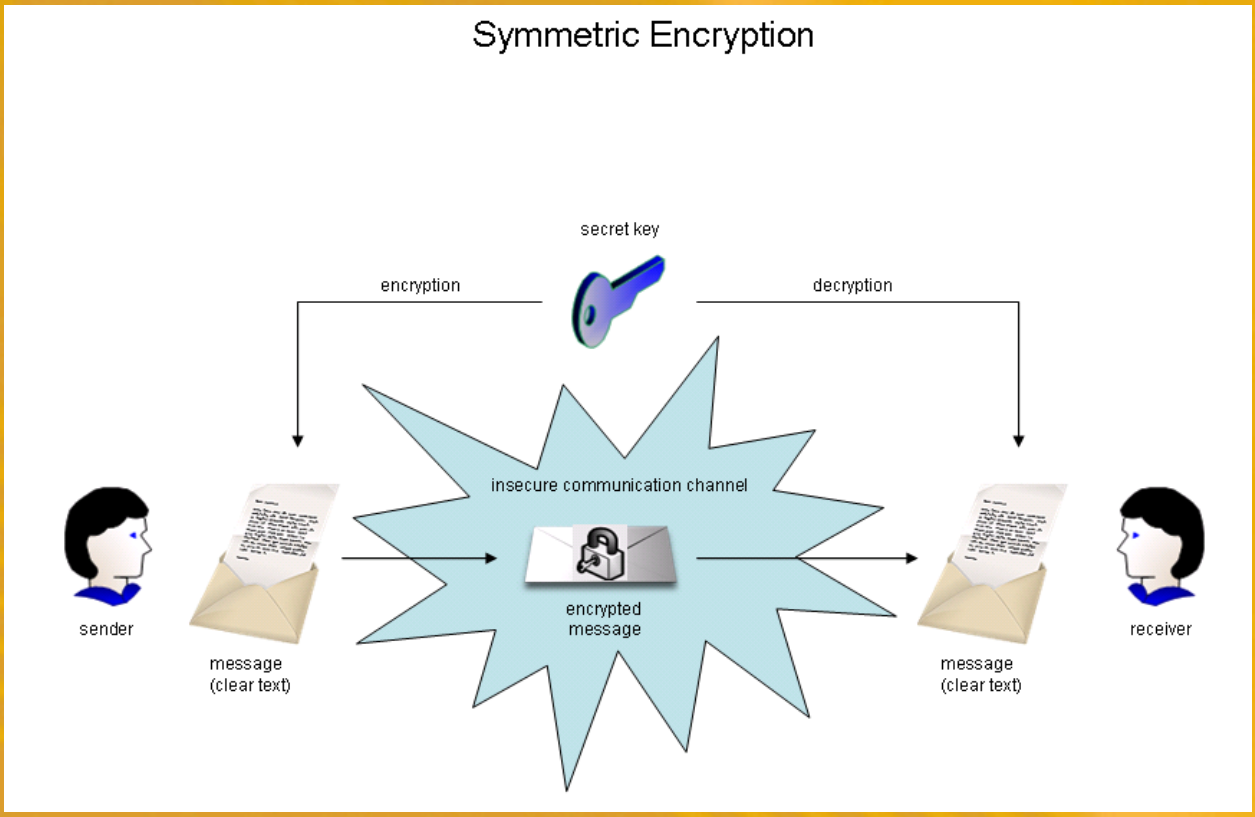

Fig.: Symmetric encryption

Example: Symmetric key (AES 256 bit)

94 d3 37 45 25 62 37 32 fd 17 3c e2 c5 29 df d2 32 22 59 9b 76 38 77 b8 d2 47 5e 64 a3 18 19 46

Large amounts of data are encrypted one block at a time. Initially, a vector with randomly generated data is fed into the encryption process (so-called *initialization vector*). The contents of this vector change during block-by-block encryption together with the secret data generated in the previous step. This prevents identical clear text blocks being mapped to identical encrypted blocks which would make it possible to draw conclusions regarding the key that was used.

Example: Initialization vector (128 bit)

c2 f9 78 df ea 20 e8 04 5b d7 94 0c 99 49 41 fa

6.1 Key exchange

The drawback of symmetric encryption is the fact that both communication partners must agree a joint, secret key in advance. This key exchange must take place over a secure channel excluding any third parties.

The actual initialization vector need not remain secret because it is generated from random numbers.

The encryption algorithm used (*cipher*) may also be in the public domain. Only the key which is used must remain secret.

6.2 Secure symmetric algorithms

Algorithm		Key size	Block size
AES-128	Advanced Encryption Standard	128 bit	128 bit
AES-192		192 bit	192 bit
AES-256		256 bit	256 bit
3DES	Triple DES	168 bit	64 bit
Blowfish	Blowfish	128..448 bit	64 bit
Twofish	Twofish	128 bit 192 bit 256 bit	128 bit

Last revised: 12/2012

6.3 Support in libraries and frameworks

6.3.1 OpenSSL

```
openssl enc -cipher

 cipher ∈ {aes128,       aes192,      aes256,
           aes-128-cbc,  aes-192-cbc, aes-256-cbc,
           bf,           bf-cbc,      blowfish,
           des3}
```

6.3.2 Java Cryptography Architecture

```
javax.crypto.Cipher  algorithm ∈ {AES, AESWrap, Blowfish, DESede, DESedeWrap}
javax.crypto.KeyGenerator                algorithm ∈ {AES, Blowfish, DESede}
java.security.AlgorithmParameters        algorithm ∈ {AES, Blowfish, DESede}
javax.crypto.SecretKeyFactory                      algorithm ∈ {AES, DESede}
```

6.3.3 .NET

Version 3.0 and later:

```
System.Security.Cryptography.AesManaged
System.Security.Cryptography.AesCryptoServiceProvider
```

Since Version 2.0:

```
System.Security.Cryptography.RijndaelManaged
System.Security.Cryptography.TripleDESCryptoServiceProvider
```

6.3.4 CryptoAPI/CAPICOM

```
CryptEncrypt()
CryptDecrypt()
CryptEncryptMessage()
CryptDecryptMessage()

ALG_ID ∈ {CALG_3DES, CALG_3DES_112,
          CALG_AES, CALG_AES_128, CALG_AES_192, CALG_AES_256}
```

6.4 Best practices

6.4.1 AES-256

AES-256 is a widely used standard; it is supported by all established frameworks and is a high-performance, cryptographically secure encryption standard.

6.4.2 Keys which are 128 bits long and longer

Encryption using less than 128 bits is weak and susceptible to brute force attacks. Symmetric keys which are at least 128 bits long must therefore be used.

6.4.3 Keys which are 256 bits long

256-bit keys make it possible to practically exclude the possibility of brute force attacks (2^{256} is equivalent to approximately 10^{76} possible key values).

6.4.4 SSL rather than symmetric encryption

A secure communication channel based on SSL 3.0/TLS 1.0 is preferable to any programmed encryption solution, both in terms of computing effort and performance.

6.4.5 Hybrid encryption rather than symmetric encryption

Hybrid encryption methods are characterized by the fact that they provide especially strong protection for keys used and should therefore always be used in preference to simple symmetric encryption (see section entitled "Hybrid encryption").

6.4.6 Key generation

A symmetric key must consist of cryptographically secure random numbers (see section entitled "Random numbers").

6.4.7 Initialization vector

All symmetric encryption must be based on an initialization vector (IV) with randomly generated content in order to protect the key which is used against brute force attacks. The IV is fed into the encryption of the first data block and changes its content with each subsequent block which is encrypted.

Initialization vectors must not be reused. Therefore a new initialization vector must be generated for every application instance or session. A new initialization vector must be generated before every encryption process in the case of data which is especially sensitive.

Initialization vectors do not need to be kept secret[2]. Nevertheless, they must only be known to the encrypting and to the decrypting instance.

6.4.8 Cipher Block Chaining (CBC)

The initialization vector on its own does not protect the key which is used against cipher text blocks which occur repeatedly. Using *Cipher Block Chaining* (CBC) achieves further, important key security. In this block mode, every clear text block is XORed one bit at a time with the preceding cipher text block before it is encrypted. This results in cipher text blocks which are always different, even if the clear text contains identical data blocks.

CBC must be used as the block mode. *Cipher Feedback* (CFB), *Cipher Text Stealing* (CTS) and *Output Feedback* (OFB) block modes are also considered to be secure but must only be used in justified exceptional cases (for instance if CBC is not available).

6.4.9 PKCS #7 padding

As a rule, the last clear text block which is to be encrypted must be padded to the block size on which the algorithm is based. This appending of characters which is referred to as "padding" must be performed using the "Public Key Cryptography Standards #7" standard. PKCS #7 is widely used and is supported by all established frameworks.

---

2 A cryptanalyst who analyzes the n cipher text blocks already knows n-1 initialization vectors when *Cipher Block Chaining* block mode is used because: The cipher text of block B(n-1) is fed into the encryption of block B(n) as the initialization vector.

6.4.10 Deriving a key from a password

If a symmetric key is derived from a password or from a pass-phrase, a cryptographically secure algorithm must be used to achieve this. *Password-Based Key Derivation Function* (PBKDF2, RFC 2898) is an especially suitable standard. *PBKDF2* adds a salt value to the password and applies a hash function, symmetric encryption and/or a Message Authentication Code to it. This process is repeated several times in order to harden the key against password cracking. Cryptographically secure keys of any length (e.g., 256 bytes) can be generated this way.

PBKDF2 is a widely used key derivation function and is used in numerous frameworks and standard implementations, for instance:

OpenSSL:	`openssl enc`
Java:	`javax.crypto.spec.PBEKeySpec`
.NET:	`System.Security.Cryptography.Rfc2898DeriveBytes`
CryptoAPI/CAPICOM:	`CryptDeriveKey(), CPDeriveKey()`

and in *WPA2*, *WinZip*, *TrueCrypt*, *OpenDocument*.

6.4.11 Strong passwords

If passwords are used as a template for symmetric keys (see *PBKDF2*), such passwords must be at least 12 characters long. A password which is less than 12 characters long can be cracked by a brute force attack within a few weeks.

6.5 Worst practices

6.5.1 DES

With its 56-bit key length, the widely used "Data Encryption Standard (DES)" algorithm is particularly susceptible to brute force attacks. The same applies to RC5.

6.5.2 Electronic Code Book (ECB)

The "Electronic Code Book (ECB)" padding mode encrypts every clear text block individually and independently of preceding blocks. Identical clear text blocks are thus mapped to identical cipher text blocks. If the clear text contains many repetitions, the key which is used can be broken by decrypting the cipher text, one block at a time, at the locations which repeat.

6.5.3 Encrypting data twice on a communication channel which has end-to-end protection

Data which is transported exclusively over a secure channel (e.g., SSL) must not be encrypted again. Duplicate encryption brings no improvement and causes loss of performance at both the sender's and the recipient's end.

Things are quite different if data packets (temporarily) leave a secure network. In this case, the data which is to be protected end-to-end must be encrypted at the sender's endpoint and decrypted at the recipient's endpoint. This serves as a defensive approach, especially if the developer has no knowledge about how the network is designed and protected.

6.5.4 Compressing encrypted data

Compressing data which has already been encrypted does not produce any significant savings in terms of the amount of data: Tried-and-tested encryption algorithms generate outputs which, from a statistical viewpoint, hardly differ from a series of random numbers. Compression algorithms, however, search for repeat patterns which almost never occur in carefully encrypted data.

On the other hand, it may be useful to compress the original data before it is encrypted: If this reduces the amount of data significantly, many blocks may provide a saving in terms of CPU cycles during subsequent encryption. Besides this, compressed data contains hardly any redundancies and this makes attacking the key even harder.

6.6 Rules of thumb

6.6.1 AES encrypts much faster than TripleDES

AES performs encryption faster than TripleDES and also offers a higher level of security (because longer keys can be selected).

6.6.2 Twofish rather than Blowfish

Although no efficient way of attacking the "Blowfish" algorithm is known at present, author Bruce Schneier recommends using its successor ("Twofish").

6.6.3 Key strength of Triple DES

Triple DES is a further development of DES and uses three 56-bit long DES keys, i.e., the total key size is 168 bits. However, because of the possibility of a man-in-the-middle attack, the effective key strength is just 112 bits.

Triple DES is, comparatively, as cryptographically secure as a 128-bit key.

6.6.4 Synchronization of encoding

In case of symmetric encryption and decryption, care must be taken to ensure that both the sender and recipient use the same character coding (e.g., UTF-8).

7 Asymmetric encryption

With asymmetric encryption, the sender and recipient of a message use different keys. There is no need for any prior exchange of keys between the two communication partners.

The keys which are used for encryption and decryption each consist of two parts: a *private key* and a *public key*. There is a fixed mathematical relationship between the two key parts but they are not interchangeable and cannot be derived from each other[3].

Such an associated key pair is referred to as a *public/private key*. The private key is secret and must not be disclosed to third parties. The public key is generally accessible and can be distributed to all potential communication partners.

Public/private key pairs are constructed on the basis of extremely large prime numbers and are practically impossible to break thanks to their mathematical complexity[4].

Example: Public Key (RSA, 1024 bit)

```
<RSAKeyValue>
  <Modulus>
    b9 d4 36 06 d0 c9 09 8b fc 6b 3b c0 59 0a 04 93 0d 97 bb b2 23
    57 77 f2 82 ac 7a ad 1a 40 2c 27 f5 b5 51 60 9f 06 f8 8d 51 1b
    b2 7c 27 05 7b 80 cb 6b bf 09 ae 71 a4 e9 8c f1 ea fd 6d 8a 85
    ca af bf a6 46 e1 b6 fe 1f bd be 2f 52 dc e5 20 ca f1 8f 66 32
    89 db 3a a8 c9 a7 d8 56 02 1a f3 e4 44 d9 1e c6 43 c2 95 4c b0
    71 61 4e 44 28 c3 b4 e7 47 82 cb d7 20 24 9a 3a 6c 8d 30 d7 9c
    0a c3
  </Modulus>
  <Exponent>AQAB</Exponent>
</RSAKeyValue>
```

[3] The public key can be derived from the private key but not the other way round.

[4] Every key of this size can be broken by deploying the requisite resources (time, computing power and expense). This is why information that is revealed by breaking a key is assumed to have become obsolete and worthless by the time it becomes known in an unauthorized manner.

Example: Associated private key (RSA, 1024 bit)

```
<RSAKeyValue>
  <Modulus>
    b9 d4 36 06 d0 c9 09 8b fc 6b 3b c0 59 0a 04 93 0d 97 bb b2 23 57
    77 f2 82 ac 7a ad 1a 40 2c 27 f5 b5 51 60 9f 06 f8 8d 51 1b b2 7c
    27 05 7b 80 cb 6b bf 09 ae 71 a4 e9 8c f1 ea fd 6d 8a 85 ca af bf
    a6 46 e1 b6 fe 1f bd be 2f 52 dc e5 20 ca f1 8f 66 32 89 db 3a a8
    c9 a7 d8 56 02 1a f3 e4 44 d9 1e c6 43 c2 95 4c b0 71 61 4e 44 28
    c3 b4 e7 47 82 cb d7 20 24 9a 3a 6c 8d 30 d7 9c 0a c3
  </Modulus>
  <Exponent>AQAB</Exponent>
  <P>
    e2 52 b1 be 4d 38 cb 0e c8 a6 b8 9e ba da 41 8c 77 0d ce 82 e2 34
    3c e3 84 1c 7e 95 c3 f4 67 9f 71 93 19 b4 99 10 6f dd c2 0a 27 96
    74 f7 f1 2c c8 d5 c4 d0 1c 7e aa c8 ae 33 37 57 a0 4a 89 77
  </P>
  <Q>
    d2 32 31 d9 5c ed 09 74 88 0c be 09 0a ed 6a bf e7 1e cb 53 76 62
    09 6e 68 5e 5a a3 79 94 58 2e 6f 97 03 0f e9 01 a8 b8 c7 bc 31 ea
    d3 65 df a8 9a 8a bd e3 b2 a7 59 ac c5 52 7a 8e 5e a8 5c 15
  </Q>
  <DP>
    7e c7 a4 19 de 58 3a 27 85 ef 1e ec 8b ef 47 58 d0 38 43 be bd c8
    55 73 7d c6 18 82 fc ca 24 62 04 d5 4a 49 08 95 54 94 fc e5 83 57
    9d 1e 67 53 97 0d 68 ba cb bb 89 1f f1 b5 6f 02 ff 1f c1 f1
  </DP>
  <DQ>
    7e c7 a4 19 de 58 3a 27 85 ef 1e ec 8b ef 47 58 d0 38 43 be bd c8
    55 73 7d c6 18 82 fc ca 24 62 04 d5 4a 49 08 95 54 94 fc e5 83 57
    9d 1e 67 53 97 0d 68 ba cb bb 89 1f f1 b5 6f 02 ff 1f c1 f1
  </DQ>
  <InverseQ>
    c2 94 bf 57 4a 79 d6 c0 7e 23 4a d4 7f ed 57 f6 19 23 f2 8f 8f eb
    c7 96 8c 17 6e 46 7c 31 35 51 f4 62 d4 77 fc 73 52 b3 67 bf 98 c4
    04 c8 99 cb 87 48 36 f4 88 55 e5 cc de 7f f7 77 46 b8 62 d4
  </InverseQ>
  <D>
    b2 a9 e7 6f d2 19 17 c6 b1 5f e0 48 e4 d2 8e e7 5b b0 40 d3 da ca
    e4 16 13 b1 05 19 9d 1d 89 d6 b9 94 24 b3 82 f2 4e d2 7f e2 e3 4e
    1e ae 54 b9 d5 da b8 07 f4 e2 6a ad e9 c7 46 29 76 c2 32 32 b3 e9
    20 f9 19 b1 c5 92 11 fc b6 e4 38 0b 27 cf f2 a0 90 df 2a b7 a3 3c
    48 d6 54 49 32 6d 67 0b 07 b8 61 73 92 e7 be 5f da f9 a1 3c 61 f2
    14 64 f6 0a 88 ba 92 20 d7 fa c4 09 7d 59 8e c9 c1 71
  </D>
</RSAKeyValue>
```

Public keys are distributed as part of a digital certificate:

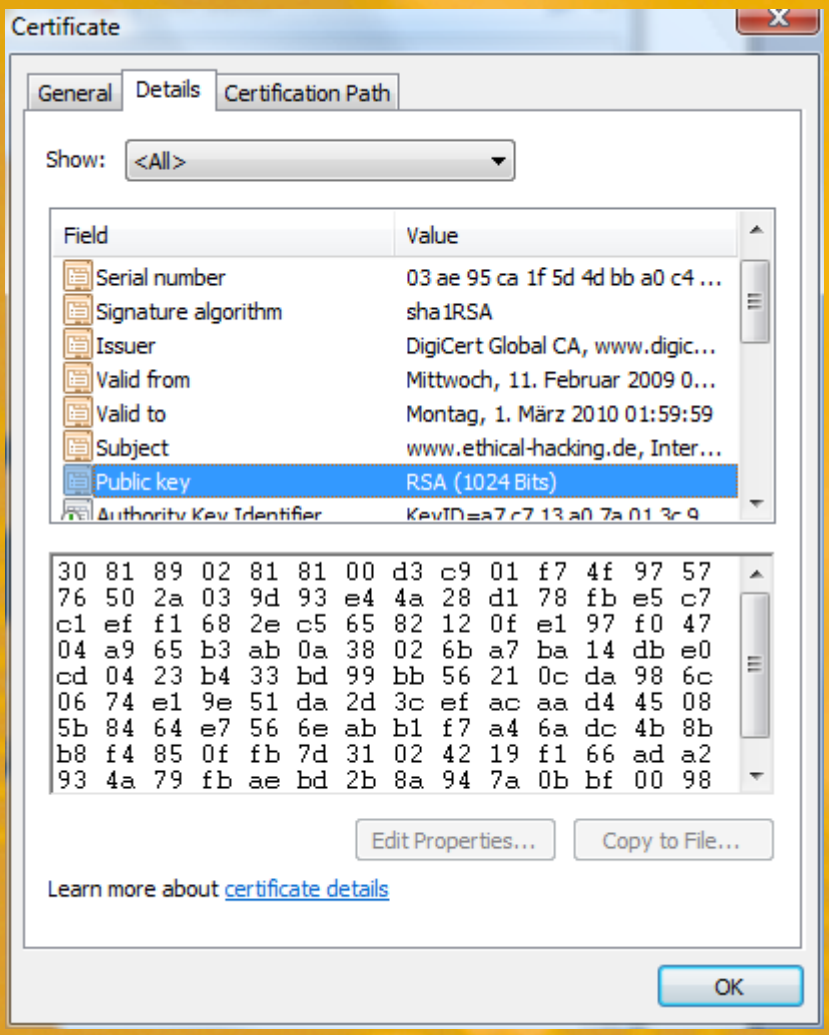

Fig.: Certificate

Each communication partner has a key pair which is unique worldwide. To send a message securely to the intended recipient, the sender encrypts the message using the recipient's public key. Only the recipient is able to decrypt the message by using his private key.

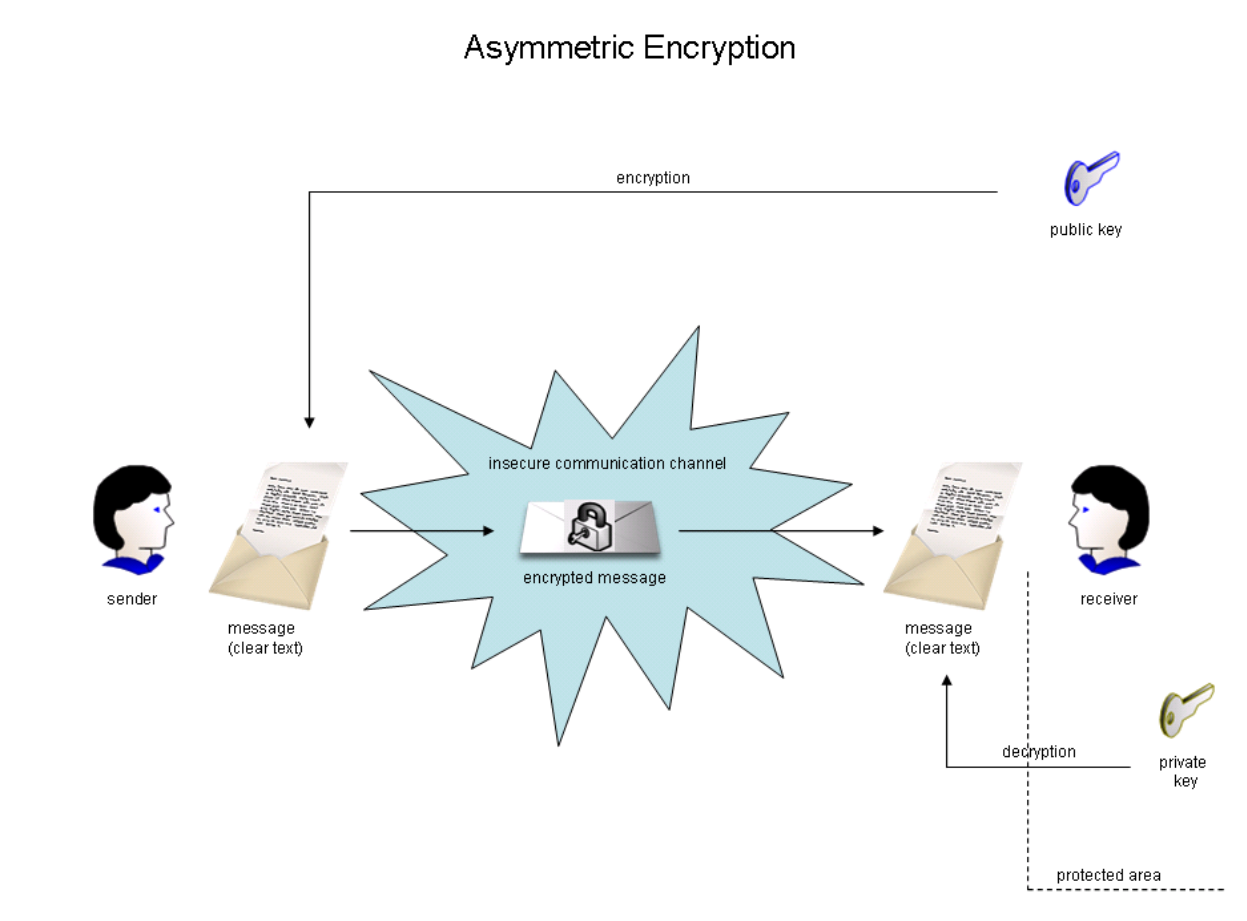

Fig.: Asymmetric encryption

7.1 Secrecy

Cryptographically secure communication based on asymmetric encryption relies on the assumption that private keys remain secret. Keys must be kept in a specially protected area of the operating system (e.g., *KeyStore* or *CertificateStore*) which only the owner of the private key can access. Ideally, the private key should be stored in hardware and be impossible to use without the explicit approval of its owner (e.g., smartcard/PIN or, in high security environments: Hardware Security Module combined with a smartcard).

7.2 Secure asymmetric encryption algorithms

Algorithm		Key size
RSA	Rivest, Shamir, Adleman	2048..16,384 bits in 8-bit increments

Last revised: 12/2012

7.3 Support in libraries and frameworks

7.3.1 OpenSSL

```
openssl rsautl
```

7.3.2 Java Cryptography Architecture

```
java.security.KeyPairGenerator      (algorithm ∈ {RSA})
java.security.KeyFactory            (algorithm ∈ {RSA})
javax.crypto.Cipher                 (algorithm ∈ {RSA})
```

7.3.3 .NET

```
System.Security.Cryptography.RSACryptoServiceProvider
```

7.3.4 CryptoAPI/CAPICOM

```
CryptEncrypt()
CryptDecrypt()
CryptEncryptMessage()
CryptDecryptMessage()

ALG_ID ∈ { CALG_RSA_SIGN }
```

7.4 Best practices

7.4.1 RSA-2048

RSA-2048 is a widely used standard: it is supported by all established frameworks and is cryptographically secure.

RSA-2048 must be used throughout the duration of the session for transmitting confidential information over a communication channel which is not secure.

Because an attacker needs computing power and time in order to break a key, RSA is used with a key size greater than 2048 (e.g., RSA-4096 or RSA-8192) with a view to long-term retention of asymmetrically encrypted information.

7.4.2 Hybrid encryption

User data should, in principle, be symmetrically encrypted because of the complex mathematical calculations which asymmetric encryption requires. The symmetric key (so-called *session key*) which is used for this is, in contrast, asymmetrically encrypted (see section entitled "Hybrid encryption").

7.4.3 Sending a message to several recipients

Only the owner of the associated private key is able to decrypt a message which has been encrypted using his public key. If a message is to be sent to several recipients, the message must be individually encrypted for each recipient (using the public key of the respective recipient).

In practice, the message is only symmetrically encrypted once, for performance reasons. The symmetric key which is used to do this is encrypted several times using the public keys of the individual recipients (see section entitled "Hybrid encryption").

7.5 Worst practices

7.5.1 RSA-1024

Because of its comparatively small key size and the threat of various attack scenarios, in practice, RSA-1024 is no longer recommended by the German Federal Office for Information Security (BSI). RSA key sizes of 2048 or more should be used instead.

7.5.2 Encrypting contents asymmetrically

Asymmetric encryption is at least 1.000 times slower than symmetric encryption. In addition, only very small data blocks can be asymmetrically encrypted.

These are reasons why data should always be symmetrically encrypted. The symmetric key which is used to do this is comparatively very small and can be asymmetrically encrypted without any significant performance loss.

8 Hybrid encryption

Hybrid encryption combines asymmetric and symmetric encryption methods. The actual data is symmetrically encrypted. Only the symmetric key which is used undergoes asymmetric encryption.

Hybrid encryption is used in practice on a large scale; one of its main applications is establishing an SSL/TLS link. The Figure below illustrates the flow of hybrid encryption:

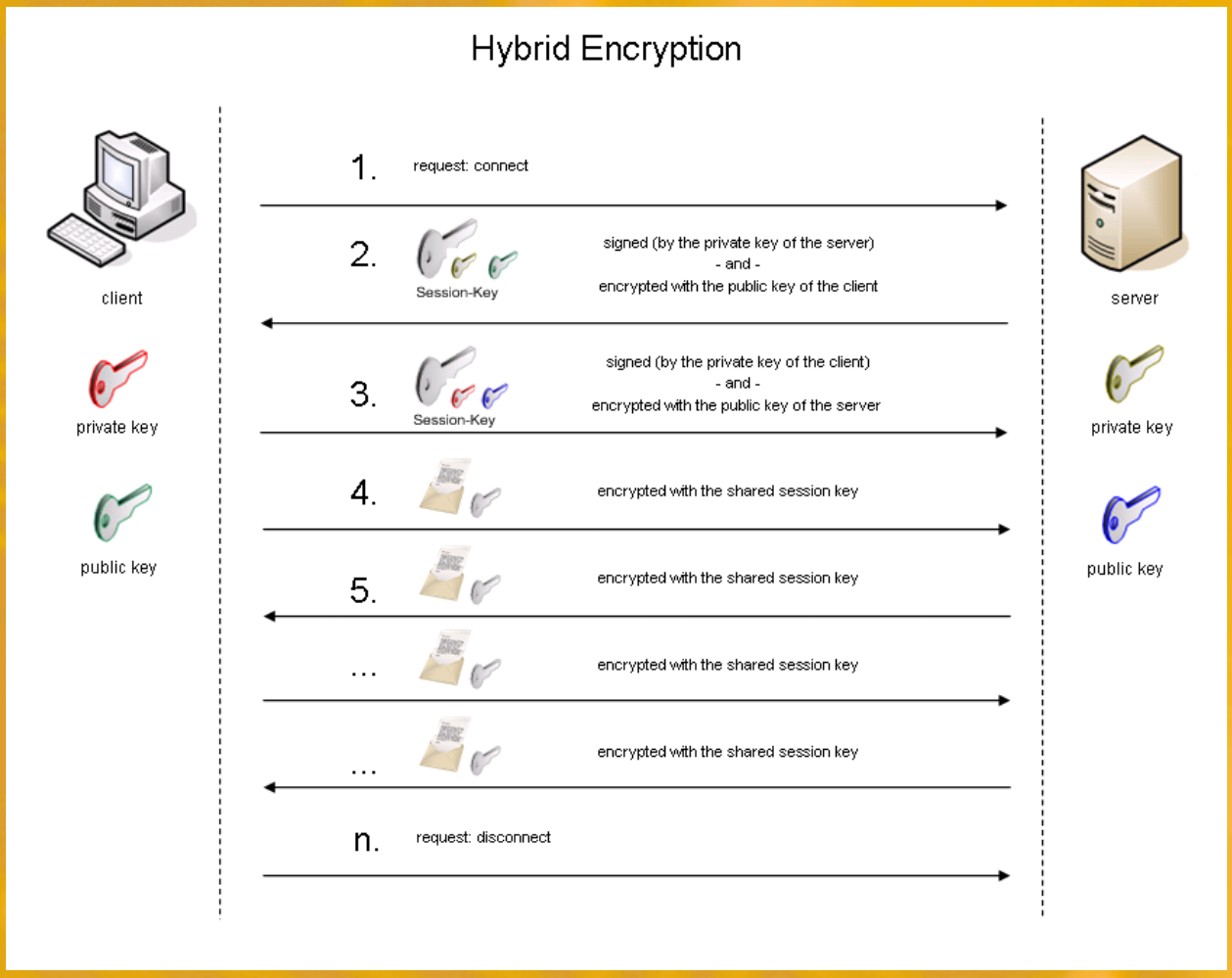

Fig.: Hybrid encryption

Explanation: Hybrid encryption in a mutual authentication procedure

When a client wants to connect to a server (step 1), the server generates a randomly determined cryptographically secure *session key*. This session key is subsequently used to symmetrically encrypt all messages which are exchanged between the client and server.

The server signs the session key so that the client can verify that the session key was generated by the server (and no one else). To ensure secure transmission to the client, the server encrypts the session key using the client's public key. The server sends the signature and the encrypted session key back to the client (step 2).

The client checks the server's signature (by using the server's public key). If the signature is valid, the server has authenticated itself to the client (*server authentication*).

The client uses its private key to decrypt the session key. From then on, the client uses this session key to symmetrically encrypt and decrypt messages that are exchanged with the server.

Next, the client must authenticate itself to the server. For this purpose, the client signs the decrypted session key and returns the signature, together with the session key, to the server (step 3); to ensure secure transmission, the key is obviously encrypted using the server's public key.

The server verifies the client's signature (by using the client's public key). If the signature is valid, the client has authenticated itself to the server (*client authentication*).

The server then uses its private key to decrypt the session key and compares this session key to the session key which it initially generated itself. If both the keys are identical, there is a secure communication link between the client and the server.

From then on, every message exchanged between the client and server is encrypted by the respective sender using the common session key and then decrypted by the respective recipient (steps 4, 5, etc.).

When the connection is subsequently cleared, both parties irrevocably delete the session key which was previously used for this session (step n). A new session key must be generated in order to set up a new connection between the two parties. The session key is therefore only valid for the duration of a session.

8.1 Achieving cryptographical objectives

Assuming that all communication partners keep their private key secret, communication which is protected this way (e.g. with AES-256 and RSA-2048) achieves all four cryptographical objectives:

Confidentiality

Any communication which takes place is encrypted.

Integrity

Messages cannot be altered en route from the sender to the recipient. A message which has been manipulated will cause a (mathematical) error when attempting to decrypt it. This applies both to user data as well as to exchanging the session key.

Authenticity

The client and server authenticate themselves by mutually signing the agreed session key.

Non-repudiation

The mathematical nature of asymmetric keys makes sure that no one other than the owner of the private key which is used for a signature can sign a message using its own identity.

8.2 Support in libraries and frameworks

8.2.1 OpenSSL

```
openssl smime -cipher            (cipher ∈ {aes128, aes192, aes256, des3})
```

8.3 Best practices

8.3.1 Fast and secure

Hybrid encryption methods exploit the comparatively high speed of symmetric encryption and the good security of asymmetric encryption.

At the same time, hybrid encryption is unaffected by the disadvantages of these individual encryption methods: Firstly, only a very small dataset is asymmetrically encrypted (the symmetric key which is used), secondly, no prior exchange of keys between the communication partners is required.

9 Digital signature

In the case of asymmetric encryption, the sender applies the recipient's public key to the message. Because of the mathematical nature of asymmetric keys, only the owner of the associated private key, i.e., the intended recipient, can decrypt the message.

In contrast, if the sender applies its private key to the message, this gives a second application area for public/private key pairs: Because only the sender can access its private key, the recipient knows that the message can only originate from that particular sender. This way, the sender has authenticated itself to the recipient. This process is referred to as a *digital signature*. The recipient can *verify* the sender's *signature* by applying the sender's public key to the communicated signature. If it is decrypted successfully, the signature is valid.

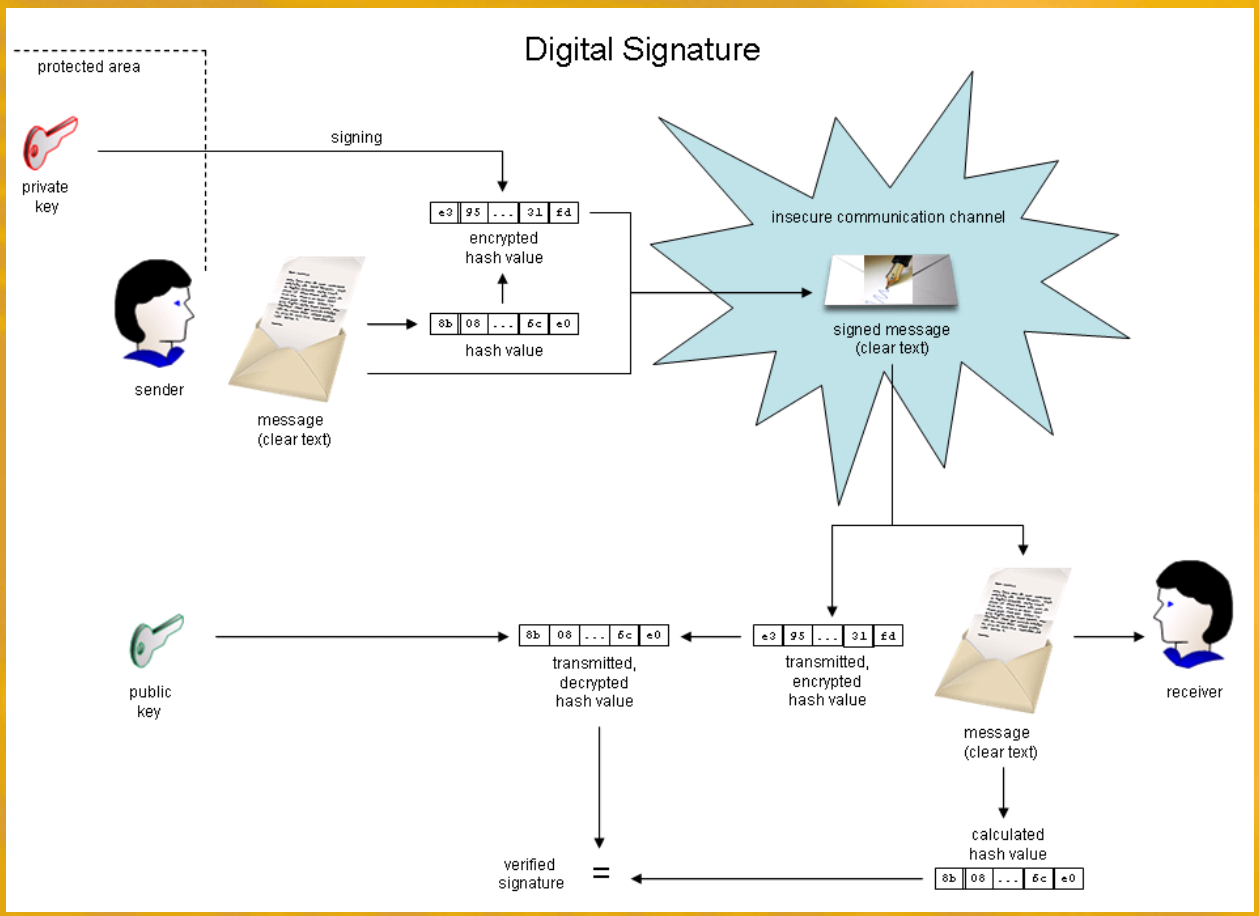

Fig.: Digital signature

In practice, the private key is not applied to the entire message. Instead, a hash value is computed for the entire message content. The signature then consists of the (small) hash value encrypted using the sender's private key. Accordingly, the recipient verifies the signature of the message by applying the sender's public key to the encrypted hash value. In order to also make sure that the message is genuine, the recipient recalculates the message's hash value and compares it to the decrypted hash value which originated from the sender (integrity check).

9.1 Secure signature algorithms

Algorithm		Key size
RSA	Rivest, Shamir, Adleman	2048..16,384 bits in 8-bit increments
DSA	Digital Signature Algorithm	1024 bit
ECDSA	Elliptic Curve Digital Signature Algorithm	256 bit 384 bit 512 bit

Last revised: 12/2012

9.2 Support in libraries and frameworks

9.2.1 OpenSSL

```
openssl dsaparam
openssl gendsa -cipher
openssl genrsa -cipher
openssl dsa
openssl rsa
openssl dgst -alg

cipher ∈ {aes128, aes192, aes256, des3}

alg    ∈ {dss1}                                  für DSA
alg    ∈ {sha256, sha384, sha512, ripemd160}  für RSA
```

9.2.2 Java Cryptography Architecture

```
java.security.KeyPairGenerator                          algorithm ∈ {RSA, DSA}
java.security.KeyFactory                                algorithm ∈ {RSA, DSA}
java.security.AlgorithmParameterGenerator                    algorithm ∈ {DSA}
java.security.AlgorithmParameters                            algorithm ∈ {DSA}
java.security.Signature            algorithm ∈ {SHA256withRSA, SHA384withRSA,
                                                    SHA512withRSA, ECDSA,
                        SHA256withECDSA, SHA384withECDSA, SHA512withECDSA}
javax.crypto.Cipher                                          algorithm ∈ {RSA}
```

9.2.3 .NET

Version 3.0 and later:

```
System.Security.Cryptography.ECDsaCng
```

Since Version 2.0:

```
System.Security.Cryptography.RSACryptoServiceProvider
System.Security.Cryptography.DSACryptoServiceProvider
```

9.2.4 CryptoAPI/CAPICOM

```
CryptSignAndEncryptMessage()
CryptDecryptAndVerifyMessageSignature()
CryptSignHash()
CryptSignMessage()
CryptSignMessageWithKey()
CryptVerifyDetachedMessageSignature()
CryptVerifyMessageSignature()
CryptVerifyMessageSignatureWithKey()
CryptVerifySignature()

ALG_ID ∈ { CALG_DSS_SIGN, CALG_ECDSA, CALG_RSA_SIGN }
```

9.3 Best practices

9.3.1 DSA rather than RSA

DSA uses a random number, so the signatures of identical data with the same key are different. However, DSA is more processor-intensive (by a factor of 10-40).

9.3.2 Signature algorithms using hash values which are 160 bits long or more

The signature of a message is represented by a hash value of the message which is encrypted using the signatory's private key.

Signature algorithms using hash values up to 128 bits long can now be attacked by using massively parallel techniques. This is why the signature algorithm must be combined with a hashing algorithm which generates hash values that are at least 160 bits long (20 bytes) (SHA-256, SHA-384, SHA-512).

9.3.3 Keeping a record of all signed documents

A digital signature is legally binding: It makes transactions impossible to repudiate in a digital world. A record of all documents that have ever been signed must be kept for use in the event of any legal disputes: Who signed a particular document, and to whom did he send it? This record must be impossible to delete (write-once read-many data medium).

10 Key exchange

In the case of symmetric encryption, both communication partners must specify a common symmetric key in advance (*key agreement*). Ideally, this key would already have been exchanged prior to communication over a separate transmission channel (e.g., face-to-face).

If the symmetric key cannot be defined until the time the connection is actually set up, this poses the problem of how the key can be securely exchanged between the two communication partners over a channel which is not yet secure.

The *Diffie-Hellman* technique offers a cryptographically secure solution. This key agreement method presupposes several parameters which need to be generated and publicly exchanged between the two communication partners in advance: A very large, randomly determined prime number p and a generator value g (typically 2 or 5).

Example: Diffie-Hellman parameters (1024 bit)

```
prime:
    00:95:3b:01:77:0f:b6:8a:ae:33:9f:ec:86:0b:99:
    93:23:bf:d4:26:95:f0:26:f7:8f:b2:68:29:6d:36:
    07:2c:27:dd:a7:60:11:71:2c:2d:d8:67:63:07:3a:
    50:5f:bb:a6:0a:a2:fb:89:6b:c2:1d:ab:1f:9e:f5:
    13:41:f3:5f:31:a5:c2:40:ea:fa:af:74:bb:95:99:
    c2:0e:01:88:1e:ce:0a:8c:e4:00:d4:04:3a:d7:bc:
    43:b5:b5:69:85:b0:50:71:60:0e:92:27:4a:33:0f:
    34:5a:c1:10:cc:f3:4c:80:16:27:50:ba:3d:db:0f:
    21:5f:c6:d8:3e:52:04:d5:0b
generator: 2 (0x2)
```

Each of the two parties can then generate an individual key pair on the basis of these two parameters: a public key and a private key. Both communication partners then exchange their public key over the communication channel which is still not secure. One partner then generate a randomly determined (symmetric) key and use the public key of the other partner to encrypt it and send it to the other partner.

10.1 Support in libraries and frameworks

10.1.1 OpenSSL

```
openssl dhparam
```

10.1.2 Java Cryptography Architecture

```
java.security.AlgorithmParameterGenerator
```

10.1.3 .NET

```
System.Security.Cryptography.ECDiffieHellmanCng
```

10.1.4 CryptoAPI/CAPICOM

```
Provider: CAPICOM_PROV_MS_DEF_DSS_DH_PROV
```

10.2 Best practices

10.2.1 Diffie-Hellman only in conjunction with mutual authentication

The problem of calculating the secret keys from the two messages in which the communication partners swap their respective secret is practically impossible to solve. But the situation is quite different in the event of a *man-in-the-middle attack*: If an attacker sits between the two communication partners before they exchange their public key, the attacker can read, modify and delete every subsequent message between the two partners without them noticing anything. This is why the Diffie-Hellman method must be used in combination with prior authentication of both communication partners (for instance by digital signatures or message authentication codes).

10.3 Rules of thumb

10.3.1 Effort involved in computing Diffie-Hellman parameters

Computing suitable Diffie-Hellman parameters requires considerable resources because a large, secure prime number must be determined. A *secure prime number* is at least 1024 bits long in the form 2p + 1 where p is in turn a prime number.

With low CPU utilization, searching for a suitable prime number takes anything from several seconds to a minute. On-the-fly calculation is therefore impossible, calculations must be performed long before a communication connection based on Diffie-Hellman key exchange is established.

11 In-memory encryption

Since Version 2.0, NET supports encryption of data in memory:

- `System.Security.SecureString`

  The *SecureString* class encrypts sensitive data and overwrites its contents in memory immediately as soon as it is no longer needed.

- `System.Security.Cryptography.ProtectedMemory`
  `System.Security.Cryptography.ProtectedData`

  The `ProtectedMemory` and `ProtectedData` classes make use of the operating-system internal *Data Protection API (DPAPI)* for encrypting sensitive data.

CryptoAPI offers corresponding operating-system internal calls with the following functions:

- `CryptProtectData()`
- `CryptUnprotectData()`
- `CryptProtectMemory()`
- `CryptUnprotectMemory()`

12 Severability clauses

12.1 Cryptographically weak algorithms

If the underlying environment, the software, framework or library which is used does not support any cryptographically secure algorithm and if, under the given circumstances, no other solution is possible, a cryptographically weak algorithm should be used rather than dispensing with protection entirely.

12.2 Commandment of caution

If algorithms which are classified as secure and recommended in this document are subsequently broken, the recommendations of the *Federal German Office for Information Security* (BSI) and the *Commandment of caution* apply until this document is next updated.

13 List of abbreviations

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
BF	Blowfish
CAPICOM	Cryptographic Application Programming Interface through Component Object Model
CBC	Cipher Block Chaining
CFB	Cipher Feedback Mode
CMAC	Cipher-based Message Authentication Code
CNG	Cryptography Next Generation
DESede	Synonym for 3DES/Triple DES
DH	Diffie-Hellman
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECB	Electronic Code Book
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
GUID	Globally Unique Identifier
HMAC	key-Hashed Message Authentication Code
IV	Initialization Vector
JCA	Java Cryptography Architecture
MAC	Message Authentication Code
MD	Message Digest
MITM	Man-in-the-Middle
OFB	Output Feedback Mode
PBKDF	Password-Based Key Derivation Function
PGP	Pretty Good Privacy
PKCS	Public Key Cryptography Standards
PKI	Public Key Infrastructure
PRNG	Pseudorandom Number Generator
RC	Rivest Cipher or Ron's Code
RIPEMD	RACE Integrity Primitives Evaluation Message Digest
RSA	Rivest Shamir Adleman
SHA	Secure Hash Algorithm
S/MIME	Secure Multipurpose Internet Mail Extension
SSL	Secure Sockets Layer
TLS	Transport Layer Security
TRNG	True Random Number Generator