# The Practical Guide to Public Key Infrastructures
# Version 1.5.0

**Copyright by**
**Nicholas Dille, nicholas@dille.name**

**Licensed under the**
**Creative Commons License**

**Published at**
**http://dille.name/docs/pkiguide.html**

Revision: 238
Last Modified: 2007-05-31 12:06

## Contents

1 Introduction....8
- 1.1 Conventions....8
- 1.2 Versioning....9
- 1.3 Localization....10
- 1.4 Nomenclature....10
- 1.5 Cryptography....10
- 1.6 Cryptographic Randomness....11

2 Symmetric Cryptography....12
- 2.1 Attacks....13
- 2.2 Well-known Ciphers....13
- 2.3 Creating Shared Secrets....13
- 2.4 Data Encryption....14
- 2.5 Summary....15

3 One-Way Functions....16
- 3.1 Checksums....16
- 3.2 Cryptographic Hash Functions....16
  - 3.2.1 Password Hashes....17
  - 3.2.2 Message Digests....18
  - 3.2.3 Message Authentication Code....18

4 Public Key Cryptography....20
- 4.1 Trapdoor Functions....20
- 4.2 Diffie-Hellman-Merkle Key Exchange....21
- 4.3 Key Schemes....21
- 4.4 Formats....22
- 4.5 Key Pairs....22
  - 4.5.1 Creating a Private Key....22
  - 4.5.2 Securing the Private Key....23
  - 4.5.3 Extracting the Public Key Data....24
  - 4.5.4 Creating a Key Pair....24
  - 4.5.5 Extracting Information from Keys....25
- 4.6 Data Encryption....27
- 4.7 Digital Signatures....28
  - 4.7.1 Signing a Message....28
  - 4.7.2 Verifying a Signed Message....30
- 4.8 Summary....30

5 Public Key Infrastructures....32
- 5.1 Certificates....32
- 5.2 Certificate Authorities....33
- 5.3 Trust Models....33
- 5.4 Certificate Chains....34
- 5.5 Verification Models....35
  - 5.5.1 Classical Verification Model....35
  - 5.5.2 Modified Verification Model....36

5.5.3 Backdating Verification Model....36
5.6 Enrolment, Renewal, Revocation....36
6 X.509 Public Key Infrastructures....38
6.1 X.500 Directories....38
6.1.1 Well-known Fields....39
6.2 X.509 Certificates....40
6.2.1 Extensions....40
6.2.2 Example....41
6.3 OpenSSL Configuration File....42
6.4 Self-signed Certificates....44
6.5 Certificate Signing Requests....46
6.5.1 Creating a CSR....47
6.5.2 Viewing a CSR....48
6.5.3 Verifying a CSR....48
6.6 Certificate Authorities....48
6.6.1 Preparation....49
6.6.2 Creating a CSR for the CA....49
6.6.3 Creating a Root CA....50
6.6.4 Creating a Subordinate CA....51
6.6.5 Enrolling a User Certificate....51
6.6.6 Embedding X.509 Extensions....52
6.6.7 Cross-signing a Certificate....53
6.6.8 Utilizing a Public Root CA....54
6.6.9 Building a Private Root CA....54
6.7 Trusted Certificates....55
6.8 Certificate Chains....55
6.8.1 Creating a Certificate Chain....56
6.8.2 Viewing Certificate Chains....56
6.9 Certificate Tasks....56
6.9.1 Verifying a Signature....56
6.9.2 Data Encryption....58
6.9.3 Verifying Signed Data....58
6.10 Certificate Revocation....58
6.10.1 Revoking a Certificate....58
6.10.2 Generating a Certificate Revocation List....59
6.10.3 Publishing a CRL....59
6.10.4 Verifying a CRL....60
6.10.5 Viewing a CRL....60
6.10.6 Status of a Certificate....61
6.11 Online Certificate Status Protocol....61
6.11.1 Requests and Responses....61
6.11.2 Responder....62
6.11.3 Miscellaneous....63
6.12 Certificate Expiry and Renewal....63
6.12.1 Checking for Certificate Expiry....63
6.12.2 Checking for Expired Certificates....64

6.12.3 Renewing a Certificate....64
6.12.4 Renewing CA Certificates....65
6.13 Conversions....65
6.13.1 Base64....66
6.13.2 Privacy Enhanced Mail....66
6.13.3 Distinguished Encoding Rules....67
6.13.4 Public Key Cryptography Standard #12....68
6.13.5 Public Key Cryptography Standard #7....68
6.14 S/MIME....69
6.14.1 Signing a Message....69
6.14.2 Verifying a Signature....70
6.14.3 Encrypting a Message....71
6.14.4 Decrypting a Message....72
6.15 Secure Socket Layer / Transport Layer Security....72
6.15.1 Server Socket....73
6.15.2 Verifying a Certificate....74
6.15.3 Client Connections....74
6.15.4 Cipher Lists....74
6.15.5 Client Certificates....76
6.15.6 Handshake....76
6.15.7 Diffie-Hellman-Merkle Parameters....77
6.15.8 Certificates for Multiple Virtual Hosts....78
6.16 Risks and Issues....79
6.16.1 Practices of the Certificate Authority....79
6.16.2 Trust Issues....80
6.16.3 Security of the Private Key....80
6.16.4 Certificate....81
6.16.5 Revocation....81
6.16.6 Non-Repudiation....82
6.16.7 Roles....82
6.16.8 Ignorance of Users....82
6.16.9 Limitation to Transport Layer....83
6.17 Summary....83
7 OpenPGP Public Key Infrastructures....84
7.1 Special Features....84
7.2 Configuration and Invocation....85
7.3 Basic Key Management....86
7.4 Backup....87
7.4.1 Owner Trust....87
7.4.2 Keys....87
7.5 Advanced Key Management....88
7.5.1 User Identities....89
7.5.2 Sub Keys....89
7.5.3 Signatures....89
7.6 Message Processing....90
7.6.1 Signatures....90

7.6.2 Encryption....91
7.7 Key Servers....92
7.8 Managing Trust....93
7.9 Summary....93
8 Comprehensive Examples....95
8.1 Common Steps....95
8.1.1 Preparations....95
8.1.2 Set up the CA....97
8.1.3 Generating Private Keys....98
8.1.4 Obtain a Certificate....98
8.1.5 Generate an Individual Secret Keys....99
8.1.6 Exchange the Certificate....100
8.2 Example 1....100
8.2.1 Ensure Data Integrity and Authentication....100
8.2.2 Ensure Confidentiality....101
8.2.3 Verify Data Integrity....101
8.3 Example 2....101
8.3.1 Exchange Secret Keys....101
8.3.2 Ensure Data Integrity and Authentication....102
8.3.3 Ensure Confidentiality....102
8.3.4 Verify Data Integrity....103
9 Concerning „Ten Risks of PKI“....104
9.1 Risk #1: Trust....104
9.2 Risk #2: Integrity of Private Keys....104
9.3 Risk #3: Integrity of the Key Store....105
9.4 Risk #4: Uniqueness of a Common Name....105
9.5 Risk #5: Authority....106
9.6 Risk #6: Impact of the User....106
9.7 Risk #7: Composite Systems....107
9.8 Risk #8....107
9.9 Risk #9: Certificate Practices....107
9.10 Risk #10: Single Sign-On....108
9.11 Summary....108
10 Summary....109
11 Appendix: Benchmarking and Sizing....110
11.1 Dissection....110
11.2 User Behaviour....111
11.3 Message Digests....111
11.4 Symmetric Encryption....112
11.5 Public Key Cryptography....112
11.6 SSL/TLS Connections....112
11.7 Interpretation....113
12 Appendix: Resolving Error Codes....114
13 Appendix: Microsoft's “makecert”....115
13.1 Creating a certificate....115
13.1.1 Saving to a File....115

13.1.2 Saving to a Certificate Store....115
13.1.3 Specifying the Subject....116
13.2 Creating a Self-Signed Certificate....116
13.3 Specifying Properties of a Certificate....116
13.4 Creating and Signing a certificate....117
13.4.1 Using a File-Based Issuer....117
13.4.2 Using a Store-Based Issuer....117
14 Appendix: Mozilla's "certutil"....118
14.1 Introduction....118
14.1.1 Creating the Database Files....118
14.1.2 Trust Attributes....118
14.2 Listing Keys and Certificates in the Database....119
14.3 Quickstart....119
14.3.1 Creating a Self-Signed CA Certificate....120
14.3.2 Creating a New Certificate....120
14.3.3 Specifying Common Properties....120
14.4 Validating a Certificate in the Database....121
14.5 Creating a Certificate Request....122
14.6 Creating a Certificate from a Request....122
14.7 Importing a Certificate into the Database....123
14.8 Deleting a Certificate from the Database....123
14.9 Modifying the Trust Attributes of a Certificate....123
14.10 Create a New Key Pair....123
14.11 Further Study....123
Bibliography / References....125
Version History....127
Roadmap....128

## Illustration Index

Illustration 1: Symmetric Cryptography....12
Illustration 2: Properties of One-Way Functions....17
Illustration 3: Public Key Cryptography....20
Illustration 4: Signatures....29
Illustration 5: Certificate....33
Illustration 6: Trust Models....34
Illustration 7: Verification of a Certificate Chain....35
Illustration 8: Classical Verification Model....35
Illustration 9: Modified Verification Model....36
Illustration 10: Backdating Verification Model....36
Illustration 11: Directories....39
Illustration 12: Self-signed Certificates....45
Illustration 13: Certificate Signing Request....46
Illustration 14: Generate a certificate from a Certificate Signing Request....52
Illustration 15: Certificate Verification....57
Illustration 16: Generate a CSR from a certificate....64
Illustration 17: OpenPGP Features....85

## Index of Tables

Table 1: Well-known Ciphers....13
Table 2: Message Digests....18
Table 3: Well-known Fields in Distinguished Names....39
Table 4: Hotkeys for openssl s_server....73
Table 5: Cipher Strings for openssl s_server and s_client....75
Table 6: Trust Attributes for certutil....119
Table 7: Certificate Usage for Validation with certutil....122

# 1 Introduction

The purpose of this document is to provide a practical introduction to Public Key Infrastructures as well as the involved fundamentals, like symmetric and asymmetric encryption and LDAP, though this document does not include a comprehensive presentation of these fundamental topics.

It is a practical guide because every aspect and tasks in this document can be explored by the included commands. Therefore, this document has a strictly technical orientation. The intended audience is composed of the intermediate and advanced user of a platform that is supported by the OpenSSL library and GNU Privacy Guard. At least some basic knowledge of computer science and/or mathematics is advantageous as well as being comfortable with a shell. The commands presented in this document are based on the Microsoft® Windows™ command prompt, though the concepts are not limited to this platform. It is assumed that Unix/Linux users are able to adapt the commands for their platform. Although this may seem discriminating, the document would contain a high fraction of redundant information if commands for Unix/Linux platforms were included as well.

The fundamentals of Public Key Infrastructures are covered in chapters 2, 3, and 4. These include an introduction to the problems that arise when using simple random number generators, how symmetric as well as asymmetric cryptography are performed and what digital signatures are.

In the central part of this document, Public Key Infrastructures are formally introduced in chapter 5 followed by a practical presentation of the concepts and tasks of X.509 PKI in chapter 6 and OpenPGP PKI in chapter 7.

Two comprehensive examples are presented in chapter 8, in which two parties are required to exchange data over a public channel utilizing a X.509 PKI. This challenge is used to demonstrate the communication between the individual elements which are involved in the process.

The document closes with a summary and the bibliography.

## 1.1 Conventions

In this document a number of mark-up conventions are used to enhance the reading experience:

- A *new phrase* is denoted by an italic font.
- A footnote[1] is used to refer the reader to a source for further study separate from this document.

1 Reference for further study

- Commands are formatted using a non-proportional font on a grey background. If necessary, long commands are broken across lines which is denoted by a backslash character on unfinished lines:

```
command 1 \
      parameter 1\
      parameter 2
command 2 \
      parameter 1 \
      parameter 2
```

- If a command, parameter or file needs to be mentioned in the text, it is formatted in a `non-proportional font`.
- The following format is used for examples that are added to support the reader in visualizing the presented concepts:

**Example:** ...

- For the reader it is often useful to be pointed to related sections or additional parameters for further study:

**Further Reading:** ...

- It often occurs that the reader needs to be aware of a connection between the current and a previous section:

**Note:** ...

- The following format is used to mark a warning:

**Warning:** ...

## *1.2 Versioning*

This document is managed by a version number consisting of three dot-separated numbers ([Major version].[Minor version].[Patch level]) and a revision number each of which represents a change of a certain severity:

- The major version number indicates a very prominent modification of the document, e.g. a new chapter or the revision of the document's structure.
- The minor version number is increased if an existing topic is extended to contain new content.
- The patch level represents the number of corrections since the last release.
- The revision number is not bound to a certain type of modification like the above. Every time the document is saved, it is incremented. Consequently, it corresponds loosely to the number of individual changes in the document.

In addition, the title page contains the date and time of the last modification. This to determine whether this document is still actively worked on.

## 1.3 Localization

This document is written and available in English only. There are no plans to localize it although German is my native language. Nevertheless some references are provided in German only because this is the original language they were published in, e.g. [22]. If a reference was originally published in English, its title is also noted, e.g. [1].

## 1.4 Nomenclature

The following list contains expressions which are vital to the understanding of this document. Before they are introduced in the corresponding technical context, an abstract meaning is established:

An **entity** is a person, an organization or a device.

An **identity** is a set of attributes that unambiguously identify an entity.

A **signature** is a means to bind information.

A **certificate** is a set of information bound together by a trusted entity.

**Confidentiality** keeps information secret from all but those who are authorized.

**Integrity** ensures that information has not been altered by unauthorized or unknown means.

**Authentication** establishes the identity of an entity.

**Authorization** is a permission to do or be something.

**Validation** is a means to provide timeliness of authorization.

**Revocation** is the retraction of certification or authorization.

**Non-repudiation** prevents the denial of previous commitments or actions.

## 1.5 Cryptography

The goal of all kinds of cryptography is to secure private information against threats of attacks by unauthorized entities. There are several layers of confidentiality:

**Organizational.** The exchange of messages is concealed by diverting another entity's attention to a seemingly important action, e.g. publicly starting an argument to conceal the exchange of the message.

**Physical.** A message is kept in a secure place, e.g. transported in a sealed envelope or written in invisible ink.

**Cryptographic.** The message is scrambled by an algorithm that requires the sender and recipient to be in possession of a secret piece of information that makes the encryption easy to perform and reverse. Without this information, it is very hard to break the encryption.

Apart from confidentiality, cryptography attempts to provide the authentication of the sender of a message so that the recipient can be reasonably sure of the identity of the sender. A by-product of the authentication is the means to allow the recipient to be confident that the message has not been modified in transit (integrity).

## 1.6 Cryptographic Randomness

Randomness is a very important requirement in cryptography because many algorithms rely on parameters that are generated in a fashion that needs to be hard to replay. Due to the fact that computers are deterministic devices, they are not able to generate sequences of truly random numbers like a *Random Number Generator (RNG)*. Therefore, they usually implement a *Pseudo-Random Number Generator (PRNG)* which relies on input that is hard or even impossible to anticipate like the time elapsed between I/O event and the contents of I/O buffers. There are also specialized devices, hardware random number generators, which are often based on the time elapsed between the particle emission during radioactive decay.

The OpenSSL library can be utilized to generate sequences of pseudo-random numbers[2]:

- Output 128 pseudo-random bits of Base64 encoded data:

```
openssl rand \
     -base64 128
```

- Write 1024 bits of pseudo-random binary data to the file random-data.bin:

```
openssl rand \
     -out random-data.bin 1024
```

This is especially useful to generate shared secrets for symmetrical cryptography. In public key cryptography, a RNG is used during the generation of the prime numbers of the private key.

2 http://www.openssl.org/docs/apps/rand.html

# 2 Symmetric Cryptography

In classical cryptography, there is a single key that is used for encryption as well as decryption (see Illustration 1). This fact makes the key a critical piece of information. Although it must be shared between the participating parties to ensure confidentiality, it cannot be made publicly available because a third party would be able to eavesdrop on the exchanged data if in possession of this key. This causes the key exchange to become a highly critical component of symmetric cryptography because it needs to be transferred securely in such a way that only the sender and the recipient are able to exchange data that is encrypted with the shared secret.

This flavour of cryptography is called symmetric because the process that is performed for encryption and decryption is equivalent. The input is processed with the shared secret to provide the output.

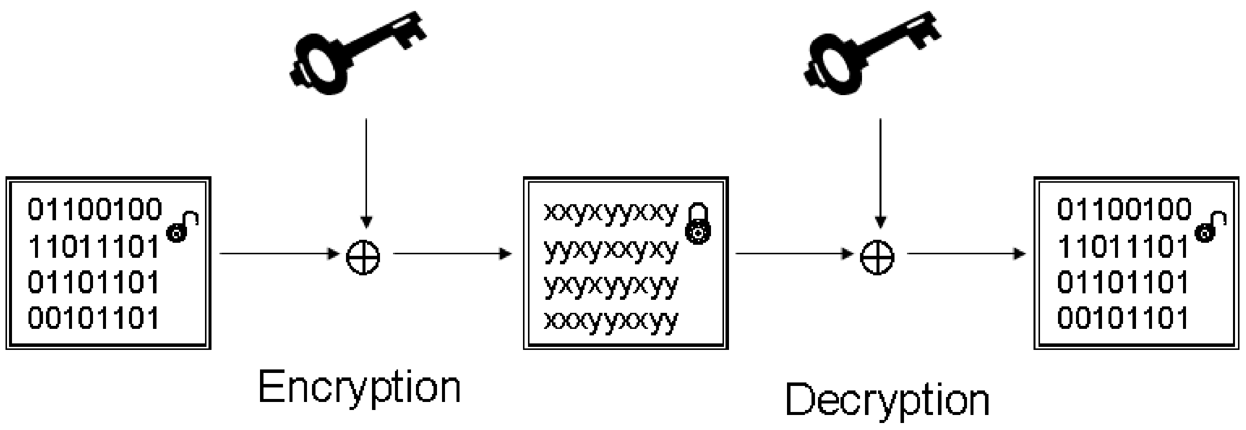

*Illustration 1: Symmetric Cryptography*

Symmetric algorithms are also called *ciphers*. The inputs for encryption to those ciphers are the *plaintext* and a *shared secret* and the output is the *ciphertext*. For decryption, a cipher accepts the ciphertext and the shared secret and returns the plaintext. Ciphers come in two flavours: *stream ciphers*, which operate on the input one bit or symbol at a time, and *block ciphers*, which split the input into equally-sized blocks of data. The latter also require padding to ensure that blocks are entirely filled with symbols. They support two modes of operation:

**Electronic Code Block (ECB).** The individual blocks are encrypted independently of each other which allows for random access to all blocks. This mode does not sufficiently change the relative frequency of symbols and, therefore, allows translation tables to be created in an easy fashion.

**Cipher Block Chaining (CBC).** The plaintext of the current block is XOR'ed with the ciphertext of the previous block to make up for the short-comings of the ECB. For the first block an *Initialization Vector (IV)* is required which is a vital part of the algorithm, because a badly chosen IV may lead to insecure cryptography.

## 2.1 Attacks

In symmetric cryptography, ciphers are especially vulnerable to the following types of attack which are performed to collect data in order to derive the shared secret.

**Ciphertext-only.** The attacker needs to collect a sufficiently high number of ciphertext messages.

**Known-plaintext.** The attacker needs to collect a sufficiently high number of plaintext messages and the corresponding ciphertext messages.

**Chosen-plaintext.** If the attacker gains access to the encryption facility without direct access to the shared secret, he is able to encrypt a collection of chosen plaintext messages.

**Chosen-ciphertext.** If the attacker gains access to the decryption facility without direct access to the shared secret, he is able to decrypt a collection of chosen ciphertext messages.

## 2.2 Well-known Ciphers

The following table lists some important properties about several well-known ciphers:

| *Cipher* | *Type* | *Block size* | *Key length* |
|---|---|---|---|
| Rivest Cipher 4 (RC4) | Stream | - | - |
| Rivest Cipher 6 (RC6) | Block | 128 bit | 128 bit,<br>192 bit,<br>256 bit |
| Data Encryption Standard (DES) | Block | 64 bit | 56 bit |
| Triple DES (3DES) | Block | 64 bit | 168 bit |
| Advanced Encryption Standard (AES) | Block | 128 bit | 128 bit,<br>192 bit,<br>256 bit |
| Blowfish | Block | 64 bit | 32-448 bit |
| Carlile Adams and Staffort Tavares (CAST) | Block | 64 bit | 40-128 bit |
| Internation Data Encryption Standard (IDEA) | Block | 64 bit | 128 bit |

*Table 1: Well-known Ciphers*

## 2.3 Creating Shared Secrets

A badly chosen shared secret can seriously reduce the security of symmetric cryptography and enable various types of attacks. Therefore, the shared secret should be created pseudo-randomly instead of being chosen deliberately. Please refer to chapter 1.4 for an introduction to cryptographic randomness and commands to generate sequences of pseudo-random numbers.

## 2.4 Data Encryption

The OpenSSL library implements several algorithms for symmetric cryptography. The cipher and its parameters are expressed by a string of dash-separated pieces of information like the name of the cipher and, if applicable, the key length and the mode of operation.

**Example:** The AES cipher with a key length of 256 bits in CBC mode is represented by the string `aes-256-cbc`.

The full list of supported ciphers is retrieved by the following command:

```
openssl list-cipher-commands
```

The following commands demonstrate the interactive usage of the AES cipher with a 256 bit key length in CBC mode by utilizing the *enc*[3] command of the OpenSSL library:

- Encrypt the file plaintext.txt and write the output to ciphertext.txt:

```
openssl enc \
     -e -aes-256-cbc \
     -in plaintext.txt -out ciphertext.txt
```

- Decrypt the file ciphertext.txt and write the output to plaintext.txt:

```
openssl enc \
     -d -aes-256-cbc \
     -in file.txt.enc -out file.txt
```

**Note:** The data is encoded using Base64 by adding the `-base64` parameter. This allows binary data to be represented by a set of 64 printable ASCII characters which enabled the transmission across channels. Some characters to provide protocol features like flow or session control.

The commands can also be executed non-interactively by adding the –pass parameter and specifying a password source on the command line:

- Specify a file containing the passphrase:

```
openssl enc \
     -e -aes-256-cbc \
     -pass file:FILE \
     -in file.txt -out file.txt.enc
```

- Specify the passphrase on the command line:

```
openssl enc \
     -e -aes-256-cbc \
     -pass pass:PASSPHRASE \
     -in file.txt -out file.txt.enc
```

**Note:** The ciphertext does not contain information about the parameters used in the encryption. Therefore, the participating parties are required to exchange this information

3 http://www.openssl.org/docs/apps/enc.html

using a common protocol. For an example of such an encoding, please refer to section 6.13.

**Note:** As demonstrated in section 1.6, the OpenSSL library can be utilized to generate a shared secret pseudo-randomly which, in general, provides a higher security than choosing one by yourself.

**Further Reading:** Explore the parameters used for encryption by specifying a custom salt (`-S SALT`) and by using a custom key (`-K KEY`) or initialization vector (`-iv IV`). All of these parameters need to be supplied with hex digits only. By adding the `-P` parameter all three value are displayed and the `enc` command terminates. In addition, the key generation is influenced by specifying a message digest algorithm (`-md MD`). Valid values are `md2`, `md5`, `sha`, and `sha1`.

## 2.5 Summary

Symmetric cryptography provides confidentiality by encrypting data using symmetric algorithms called ciphers. Due to their short keys and the moderate complexity of the involved mathematical operations, they provide a high throughput and allow for fast bulk encryption of data. Ciphers can also be used to construct stronger ciphers, e.g. by chaining them. Their weakness lies in the need for a shared secret which must be exchanged over a secure and trusted channel.

# 3 One-Way Functions

In modern cryptography, *one-way functions* are a fundamental concept. Such a function takes a piece of data as input and computes a piece of output. To break the algorithm an attacker would have to reverse the function, meaning that the original input data needs to be computed from the output data, but this is computationally expensive. This is also called the *one-way* property of a function.

Although it has not been proven that one-way functions really exist, it is expected that such functions are computationally hard to reverse. It was shown that one-way functions exist if the central problem of computer science is proven: $P \neq NP$.

One-way functions are also called *hash functions* which produce a hash for each piece of input.

## 3.1 Checksums

When a message is transmitted electronically, it may get modified by interference or fading. Since this modification cannot be prevented, messages are augmented with redundant data, so-called *checksums* that allow for such modifications to be detected or even corrected.

Although such checksums are the result of one-way functions, a second property is more important. If a message with a checksum suffers an extensive modification, it may cause a modification to be detectable but the message to be unrecoverable, or it may even cause a modification to be undetectable. To prevent the former case, a checksum needs to be *collision-resistant*[4], meaning that it is unlikely to produce a message that reduces to the same checksum and, therefore, cause a collision. Such a checksum enables the recipient to verify the integrity of the received message with a certain degree of confidence.

Simple checksums, like the *parity*, may even become corrupted themselves in such a way that an unmodified message is correctly considered to be unmodified although the parity has been modified. Although this does not seem to be critical, it shows that the underlying one-way function is able to assign to each message more than one checksum. This makes the checksum less effective.

## 3.2 Cryptographic Hash Functions

In modern cryptography, one-way functions are required for public key cryptography as well as digital signatures. Apart from the one-way property, digital signatures are required to be collision-resistant but not in the sense of checksums. Instead of being unlikely to generate a second message which reduces to the same hash, a cryptographic hash function needs to produce hashes in such a way that it is computationally hard to find a message with the same hash.

4 In computer science, this property is expressed by the Hamming distance which measures the required extent of a modification for it to cause a collision.

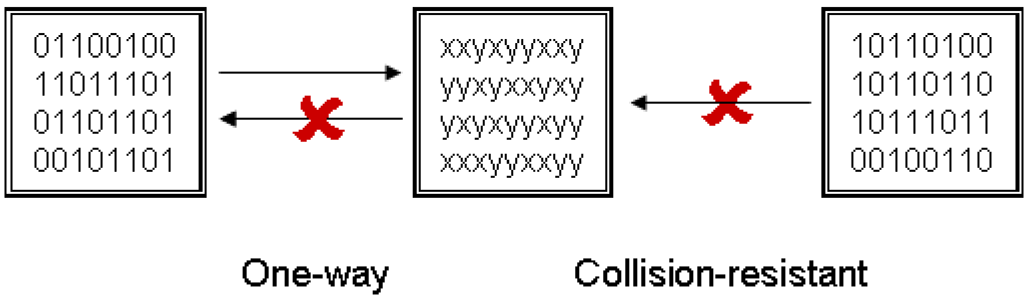

*Illustration 2: Properties of One-Way Functions*

### 3.2.1 Password Hashes

Hash functions are commonly used to store and transmit passwords because the original password cannot be extracted from an intercepted password hash in an efficient manner. With the knowledge of the applied hash function an attack can use an intercepted hash for dictionary based and brute force attacks. For both types of attacks, strings are chosen in a more or less intelligent manner, hashed using the appropriate algorithm and compared to the intercepted password hash. A dictionary based attack assumes that most passwords are either found in a dictionary or are easily derived from one or more words found in a dictionary. A brute force attack creates every possible permutation of a set of characters for typical (or even all permitted) password lengths and searches for the same hash.

The following command creates a crypt-style hash of 11 bytes from a password and a salt of 2 bytes which is stored with the password and used to randomize the result. The salt can also be specified on the command line to enforce a preset salt[5]:

```
openssl passwd \
     -salt 8E MySecret
```

Due to shortcomings of crypt-style password hashes, modern systems use MD5 based password hashes. One reason is that MD5 uses longer salts (8 bytes) and produces longer hashes (22 bytes) which reduce the chance of accidentally creating a collision.

```
openssl passwd \
     -1 \
     -salt sXiKzkus MySecret
```

MD5 based hashes also use a different format which allows both password hashes to be used simultaneously.

**Further Reading:** For additional insight, please refer to "*Customers, Passwords, and Web Sites*" by Bruce Schneier.

The parameter `-apr1` allows a variant of the MD5 based password hashes to be used which was developed by the Apache Group[6].

5 http://www.openssl.org/docs/apps/passwd.html
6 http://www.apache.org

### 3.2.2 Message Digests

Whereas a checksum provides some protection against modifications during electronic transmissions, a *message digest* (or *digest*) is a one-way function which secures a message against tampering. These functions reduce the input, a message of variable length, to a hash value of a fixed length.

Well-known message digests are described in the following table:

| ***Message Digest*** | ***Hash size*** | ***Comments*** |
|---|---|---|
| Message Digest 5 (MD5) | 128 bit | Was shown not to be collision-resistant |
| Secure Hash Algorithm (SHA) | Depends on variant | Successor to MD5<br>Variants include: SHA-1 (160 bit), SHA-256 (256 bit), SHA-512 (512 bit) |

*Table 2: Message Digests*

A list of message digests supported by the OpenSSL library is retrieved by the following command:

```
openssl \
     list-message-digest-commands
```

A message which is transmitted with a digest can be verified to be received untampered. The recipient calculates the digest and compares it with the one included in the message. This does not prevent an attacker to substitute the digest after modifying the original message.

**Warning:** In *„Cryptoanalysis of MD5 and SHA1: Time for a New Standard*" by Bruce Schneier, the author recommends to use later versions of the SHA message digest algorithm because MD5 was shown not to be collision-resistant and SHA-1 is rumoured to be vulnerable.

### 3.2.3 Message Authentication Code

A message digest does not prevent an attacker from modifying the original message and adding a new message digest because the recipient cannot differentiate between the original and the modified message. For a sender and recipient who are sharing a secret key, the message can be transmitted with a *message authentication code (MAC)*, a symmetrically encrypted message digest using a shared secret. A successfully verified MAC assures the recipient of the integrity and the authenticity of the message because it must have originated from an entity in possession of the shared secret.

A special kind of MAC is the *hash-keyed message authentication code (HMAC)*[7] which also involves a shared secret that is not used for encryption. The shared secret is appended to the message and both are hashed to generate the message digest. Although an attack can modify the message and exchange the digest, the recipient notices this because the attacker does not know the shared secret so that the resulting message digest is recognized to be invalid.

7 http://www.ietf.org/rfc/rfc2104.txt

A message authentication code does not provide the same security a digital signature because it can be created by every entity in possession of the shared secret. An introduction to digital signatures is provided in section 4.7.

# 4 Public Key Cryptography

In symmetric cryptography, a high organizational overhead is caused by the exchange of shared secrets. It is necessary to utilize a secure channel[8] or an encrypted public channel. If the shared secret is intercepted, the confidentiality of the exchanged data is compromised. In *public key cryptography*, the need for a shared secret is eliminated. Instead, each party possesses a *key pair* consisting of a *private key* and a *public key*. After generating ciphertext with one of the two keys only the other key is able to reverse the encryption (see Illustration 3) which makes public key cryptography *asymmetric*.

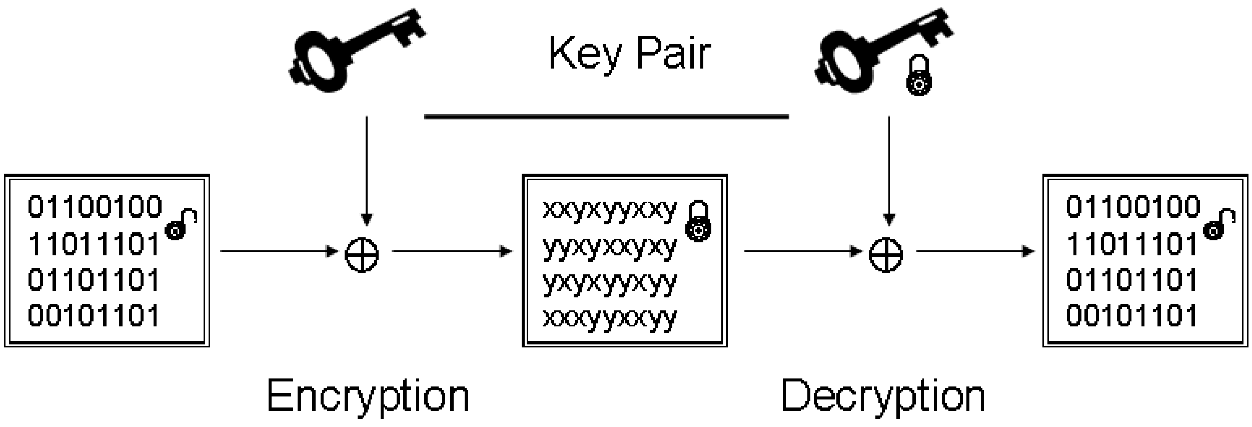

*Illustration 3: Public Key Cryptography*

In addition to confidentiality, public key cryptography allows for important properties in private communication using public channels: the *authentication* of the participating parties and the *integrity* of the exchanged data. Confidentiality is achieved by encrypting data using the public key of the recipient, so that only he can decrypt the data because he is in possession of the private key. Authentication requires signing the exchanged data with the sender's private key which allows everyone in possession of the sender's public key to verify the sender's identity. Data integrity is covered in a later in this chapter.

While it is encouraged to distribute the public key via a publicly available service like a directory, the private key is a critical piece of information because it can be misused to forge authentication data and eavesdrop on private communication.

## 4.1 Trapdoor Functions

Public key cryptography is based on a special flavour of one-way functions which have the trapdoor property and are also called *trapdoor functions*. Such a function is irreversible[9] without the knowledge of some secret information. When the public key is used to asymmetrically encrypt a message, this is considered a trapdoor function because the original message can only be recovered with the private key which represents the secret knowledge to reverse the encryption.

8 ... or rather a channel that is assumed to be secure ...
9 ... or rather computationally hard to reverse ...

## 4.2 Diffie-Hellman-Merkle Key Exchange

The Diffie-Hellman-Merkle (DH)[10] key exchange algorithm allows two parties to establish a shared secret via a public channel without prior common knowledge. This protocol allows the foreign entities to engage in private communication without the need to exchange a shared secret which would create an obstacle hardly possible to overcome. It also provides the mathematical foundation for public key cryptography.

The DH key exchange is based on a trapdoor function which requires both parties to generate private trapdoor information and public values. The latter is exchanged on the public channel. The recipient's private information is used with the sender's public information to calculate the shared secret.

**Note:** Although an eavesdropper cannot compute the shared secret or the trapdoor information in a timely manner, the protocol is still susceptible to *Man-in-the-Middle (MitM) attacks* because the eavesdropper could intercept the exchanged data and substitute it with locally generated information. Consequently, both parties assume that the key exchange is performed with the expected entity but, instead, the eavesdropper swaps the parameters for the key exchange protocol and, later, forwards, modifies or, even, forges data that is sent to one or both parties in the name of the other.

The parameters that are required for the Diffie-Hellman-Merkle key exchange are pre-generated by the following command[11]:

```
openssl dhparam \
     -out dh.pem 1024
```

The resulting parameters are displayed by the following command:

```
openssl dhparam \
     -text -noout \
     -in dh.pem
```

To utilize the DH parameters, an example will be presented in a later chapter which demonstrates SSL connections (see chapter 6.15.7).

## 4.3 Key Schemes

There are several key schemes which are based on different mathematical problems that are considered or proven to be computationally hard to solve. Therefore, breaking the encryption provided by public key cryptography is equivalent to solving a hard mathematical problem in a computationally efficient manner.

The most popular key scheme is RSA by Ron Rivest, Adi Shamir and Len Adleman. The underlying problem is based on the power function which is solved by efficiently factoring products of very large primes. Although solving this problem is considered to be computationally hard, this has not been proven mathematically.

10 Most readers will know of this algorithm simply by the name of Diffie-Hellman but Hellman proposed that Merkle is mentioned in its name because of his contribution to the invention of public key cryptography.
11 http://www.openssl.org/docs/apps/dhparam.html

The second supported key scheme is the Digital Signature Algorithm (DSA) which is based on the discrete logarithm problem[12] and provides only a signature algorithm. It has been proven to be hard to solve making this scheme more secure but also more complex. Its complexity causes DSA to show a lower performance than RSA.

## 4.4 Formats

The private and the public key are expressed using the *Abstract Syntax Notation (ASN.1)*[13], a standardized notation for data structures created by the ISO and ITU-T. The resulting notation is unambiguous and independent of machine or system specific encodings. Data structures can be expressed by several sets of encoding rules including the *Distinguished Encoding Rules (DER)* commonly used on Microsoft® Windows™ based systems.

The OpenSSL library uses DER and Base64 formatted DER encoded keys because the resulting data does not contain ASCII characters which may be interpreted as session for flow control commands by networking components.

**Further Reading:** The corresponding ASN.1 structure can be extracted and displayed by the asn1parse command of the OpenSSL library.

**Further Reading:** In section 6.13 several other formats are discussed and commonly used conversions are presented.

## 4.5 Key Pairs

This section explains the creation of key pairs consisting of a private and a public key which are suitable for use in the RSA and DSA schemes of public key cryptography.

In addition, commands demonstrate how a clear text representation can be extracted from the individual keys and how they can be combined to a key pair.

### 4.5.1 Creating a Private Key

After creation a private key contains all necessary data for both keys so that a public key can be restored from the private key. This makes the private key even more important so that its privacy must be protected at all costs.

A private key can be generated for both supported key schemes, RSA and DSA, and symmetrically encrypted with different key lengths. For the sake of simplicity, the following commands demonstrate the creation of a 1024 bit RSA and DSA private key which is saved to a file without being protected by a passphrase.

- Generate a 1024 bit RSA based private key[14]:

12 In fact, the DSA scheme is based on the ElGamal public key algorithm which was developed by Taher Elgamal and is built on the discrete logarithm problem.
13 http://www.asn1.org/
14 http://www.openssl.org/docs/apps/genrsa.html

```
openssl genrsa \
     -out host_rsa_key.pem 1024
```

- Generate a 1024 bit DSA based private key[15]:

```
openssl dsaparam -genkey \
     -out host_dsa_key.pem 1024
```

- There is a variant to generate DSA based private keys based on a DSA parameter file. After creating such a parameter file several DSA based private keys can be bulk-generated[16]:

```
openssl dsaparam \
     -out host_dsa_prm.pem 1024

openssl gendsa \
     -out host_dsa_key1.pem host_dsa_prm.pem

openssl gendsa \
     -out host_dsa_key2.pem host_dsa_prm.pem
```

**Note:** In the above commands, the DSA private keys are generated from the same set of parameters. This causes the set of private keys to share some values and differ only in their dedicated private and public component.

Although the presented commands did not demonstrate this, it is strongly encouraged to protect the private key with a passphrase. This is achieved by specifying a cipher algorithm on the command line, e.g.:

```
openssl genrsa \
     -aes256 \
     -out host_rsa_key.pem 1024
```

You will be asked to supply a passphrase before the private key is stored in the file. Similar to the passphrase expected for symmetric encryption (see section 4.6), the passphrase for the private key can be provided using the `-passout` parameter. Subsequent key operations automatically require the passphrase to be entered or to be provided on the command line using the `-passin` parameter.

### 4.5.2 Securing the Private Key

The previously generated private keys can also be rewritten to be protected by a passphrase. The following command adds a new passphrase or changes a previously added passphrase[17]:

```
openssl rsa \
     -in host_rsa_key.pem \
     -aes256 -out host_rsa_key.pem
```

Although it is not recommended, the passphrase can be removed from a private key file using the following command:

15 http://www.openssl.org/docs/apps/dsaparam.html
16 http://www.openssl.org/docs/apps/gendsa.html
17 http://www.openssl.org/docs/apps/rsa.html

```
openssl rsa \
     -in host_rsa_key.pem \
     -out host_rsa_key.pem
```

A corresponding OpenSSL command exists for DSA based private keys to add or change a passphrase[18] …

```
openssl dsa \
     -in host_rsa_key.pem \
     -aes256 -out host_rsa_key.enc
```

… and remove the passphrase:

```
openssl dsa \
     -in host_rsa_key.enc \
     -out host_rsa_key.pem
```

**Note:** Apart from the protection by a passphrase, a private key should be placed in a secure directory and be included in regular backups.

**Note:** Some of the presented commands require a passphrase to be entered. Non-interactive or unattended variants can be built by using the `-passin` and `-passout` parameters for reading and writing the private key, respectively. See section 4.5.2 for details.

### 4.5.3 Extracting the Public Key Data

The public key consists of a subset of information that is contained in the private key file. It is extracted from a RSA based private key by the following command:

```
openssl rsa \
     -pubin -in host_rsa_key.pem \
     -out host_rsa_pub.pem
```

The DSA based public key consists of a single value, the modulus, which allow for public key operations to be performed. It is extracted from a DSA based private key by the following command:

```
openssl dsa \
     -modulus -in host_dsa_key.pem \
     -out host_dsa_pub.pem
```

**Note:** The public fraction of a key pair cannot be protected by a passphrase because it is intended to be shared publicly.

### 4.5.4 Creating a Key Pair

Instead of individual key files the owner may create a key file that contains both parts. This allows for easier use because, in many commands, it is not necessary to distinguish between the individual keys.

The following commands combine the corresponding RSA based private and public keys:

18 http://www.openssl.org/docs/apps/dsa.html

```
openssl rsa \
     -in host_rsa_key.pem >host_rsa.pem
openssl rsa
     -pubout \
     -in host_rsa_key.pem >>host_rsa.pem
```

**Note:** The private key contained in this key pair file should be protected in the same manner as explained in section 4.5.2.

To separate the individual keys, only the private key is utilized:

```
openssl rsa \
     -in host_rsa.pem -out host_rsa_key.pem
openssl rsa \
     -pubout \
     -in host_rsa.pem -out host_rsa_pub.pem
```

**Note:** Analogous commands based on the dsa command are used to combine and separate DSA based keys in a single PEM formatted file.

### 4.5.5 Extracting Information from Keys

Each key contains information that can be extracted and displayed. The following commands demonstrate this for the individual key files:

- Extract a human-readable form of the RSA based private key:

```
openssl rsa \
     -text -noout \
     -in host_rsa_key.pem
```

- Extract a human-readable form of the RSA based public key:

```
openssl rsa \
     -text -noout \
     -pubin -in host_rsa_pub.pem
```

**Further Reading:** Compare the extracted information to determine which fields are shared among the keys and verify that the public key is a subset of the private key.

- Extract a human-readable form of the DSA based parameter file:

```
openssl dsaparam \
     -text -noout \
     -in host_dsa_prm.pem
```

- Extract a human-readable form of the DSA based private key:

```
openssl dsa \
     -text -noout \
     -in host_dsa_key.pem
```

- Extract a human-readable form of the DSA based public key:

```
openssl dsa \
     -text -noout \
     -pubin -in host_dsa_pub.pem
```

The RSA based public key consists of the `modulus` and the `publicExponent` fields of the private key. The following output shows a RSA private and public key in human-readable form. In the output all sequences of hexadecimal numbers are substituted with compressed zeros.

- RSA based private key:

```
Private-Key: (512 bit)
modulus:
    00:...:00
publicExponent: 65537 (0x10001)
privateExponent:
    00:...:00
prime1:
    00:...:00
prime2:
    00:...:00
exponent1:
    00:...:00
exponent2:
    00:...:00
coefficient:
    00:...:00
```

- The corresponding RSA based public key:

```
Modulus (512 bit):
    00:...:00
Exponent: 65537 (0x10001)
```

**Further Reading:** The DAS based public key consists of the pub field of the private key. The following output shows a DSA private and public key in human-readable form.

- DSA based parameters:

```
DSA-Parameters: (512 bit)
    p:
        00:...:00
    q:
        00:...:00
    g:
        00:...:00
```

- DSA based private key:

```
Private-Key: (512 bit)
priv:
    00:...:00
pub:
    00:...:00
P:
    00:...:00
Q:
    00:...:00
G:
    00:...:00
```

- The corresponding DSA based public key:

```
Public Key=00...00
```

**Further Reading:** Private as well as public keys can also be displayed by the ASN.1 parser provided by the OpenSSL library[19]. See section 4.4 for an introduction.

## 4.6 Data Encryption

In public key cryptography, it is assumed that the public key is shared whereas the private key stored securely and protected by a passphrase as well as Access Control Lists (ACLs)[20]. Therefore, encryption is only supported with the public key of the recipient[21]:

```
openssl rsautl \
     -encrypt \
     -pubin -inkey host_rsa_pub.pem \
     -in plaintext.txt -out ciphertext.txt
```

The corresponding private key is used to reverse the encryption and retrieve the plaintext data:

```
openssl rsautl \
     -decrypt \
     -inkey host_rsa_key.pem \
     -in ciphertext.txt -out plaintext.txt
```

19 http://www.openssl.org/docs/apps/asn1parse.html
20 http://en.wikipedia.org/wiki/Access_control_list
21 http://www.openssl.org/docs/apps/rsautl.html

## *4.7 Digital Signatures*

So far, public key *cryptography* only replaces symmetric cryptography concerning confidentiality but does not add new features like authentication and integrity. Those features are covered by digital signatures being a signed message digest. Contrary to the encryption, digital signatures are created by encrypting a message digest with the sender’s private key in order for the recipient to be able to verify the signature using the sender’s public key.

Public key cryptography is inherently unsuited to encrypt large amounts of data. In comparison with symmetric cryptography, the throughput is reduces due to its complex mathematical operations. Therefore, it is only implemented to encrypt small amount of data. A message digest is used to produce a fixed length representation of the original data which is then encrypted using the sender’s private key. The result is called a *digital signature* and is sent along with the original message. This enables every entity in possession of the sender’s public key to verify the message digest.

In addition to confidentiality, public key cryptography allows for two additional features provided by digital signatures:

**Authentication.** The recipient can be confident about the sender’s claim of identity because only the sender was able to produce the signature with his private key.

**Integrity.** Both parties can be confident that the message is received untampered because a modification would result in an invalid signature.

The concept of digital signatures was developed to provide a digital counterpart to the traditional handwritten signature. The latter is a graphical representation of an entity’s name which is expected to be unique and hard to forge. Unfortunately, this handwritten signature can either be reproduced by skilful individuals or be transferred to another document. A digital signature can only be forged by an entity which is in possession of the sender’s private key.

### 4.7.1 Signing a Message

To assure the recipient of the authenticity of the sender or his message, the sender needs to create a digest from his message which he then encrypts with his private key (see Illustration 4). This process renders the message and the signature impossible to modify without the recipient detecting the change[22].

22 The signature is secured against tampering as long as the sender’s private key and the involved algorithms remain uncompromised.

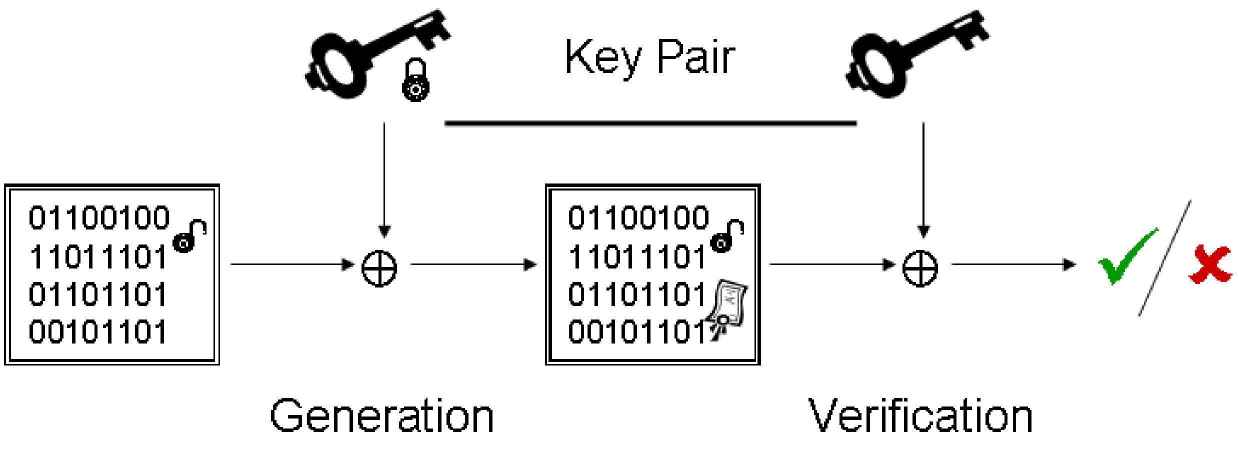

*Illustration 4: Signatures*

The following command generates a SHA1 message digest and encrypts it using the private key contained in `host_rsa_key.pem`:

```
openssl dgst \
     -sign host_rsa_key.pem \
     -sha1 \
     -out plaintext.txt.sha1 plaintext.txt
```

The same result is obtained by using the `rsautl` command:

```
openssl rsautl \
     -sign \
     -inkey host_rsa_key.pem \
     -in plaintext.txt -out plaintext.txt.sha1
```

According to the manual pages of the `dgst`[23] command, the `dss1` digest needs to be utilized when implementing DSA keys. Unfortunately, this digest is not documented in the online help provided by the OpenSSL library.

```
openssl dgst \
     -sign host_dsa_key.pem \
     -dss1 \
     -out plaintext.txt.dss1 plaintext.txt
```

**Note:** The *Digital Signature Standard (DSS)*[24] is a standard for digital signatures developed by the *National Institute of Standards and Technology (NIST)*[25] and published by the U.S. Federal government in the *Federal Information Processing Standards (FIPS)*[26].

**Note:** The `rsautl` command does not provide an alternative to the dgst command when creating a signature using DSA based keys like its RSA based counterpart.

23 http://www.openssl.org/docs/apps/dgst.html
24 http://www.itl.nist.gov/fipspubs/fip186.htm
25 http://www.nist.gov/
26 http://www.itl.nist.gov/fipspubs

### 4.7.2 Verifying a Signed Message

A signed message consists of an encrypted message digest and the message itself. Due to the fact that the message digest was encrypted with the sender's private key, every entity in possession of the sender's public key is able to verify the signature.

In the first step, the message digest is decrypted using the sender's public key. Then, a second message digest is created using the same algorithm and the same parameters as for the received digest. After successfully comparing the decrypted and the newly generated digests, the recipient can be confident that the sender's claim to be the signer can be trusted and that the received message was unmodified.

These steps can be performed by the following commands if the caller is in possession of the sender's public key:

- Verify a SHA-1 and RSA based signature:

  ```
  openssl dgst \
       -verify host_rsa_pub.pem \
       -sha1 \
       -signature plaintext.txt.sha1 plaintext.txt
  ```

  An alternative to the above command is provided by the `rsautl` command:

  ```
  openssl rsautl \
       -verify \
       -pubin -inkey host_rsa_pub.pem \
       -in plaintext.txt.sha1 -out plaintext.txt
  ```

- Verify a DSS1 and DSA based signature:

  ```
  openssl dgst \
       -verify host_dsa_pub.pem \
       -dss1 \
       -signature plaintext.txt.dss1 plaintext.txt
  ```

Apart from ensuring the integrity of a message, the signed message digest also allows for the authentication of the sender.

## *4.8 Summary*

Public key cryptography provides strong encryption and signatures. The resulting messages may be publicly shared without worrying that it might be compromised without possession of the involved private key. Longer keys are required than for symmetric cryptography and the involved mathematical operations are more complex making these algorithms unsuitable for the encryption of bulk data.

Compared to symmetric cryptography, the exchange of the public key is not as critical as the shared secret because the security of public key cryptography is not threatened by intercepting the transmission of the public key. Instead, both parties need to make sure that the public key received by the recipient is the same as offered by the sender. Consequently, the transmission of a public key does not need to be performed confidentially but, rather, be performed after authenticating the sending party and securing the integrity of the public key.

**Note:** Compared to symmetric cryptography, public key cryptography is a very young area of research. Due to this fact, algorithms have not been analyzed as thoroughly as the ciphers in symmetric cryptography. In addition, some algorithms are based on mathematical problems which are expected but not proven to be hard to solve, i.e. an efficient solution for the involved problems remains unknown at the time of this writing.

# 5 Public Key Infrastructures

In symmetric cryptography, the transmission of the key, the shared secret, was critical because it must be kept confidential. Public key cryptography allows public channels to be used for the key exchange because it does not contain sensitive information. But a public key needs to be obtained from a trusted source to be confident that it was received unmodified. Although this is an improvement, the source of a public key needs to be authenticated and trusted which causes some overhead for all participating entities.

A *Public Key Infrastructure (PKI)* is based on public key cryptography and allows for some entities to take on the special role, called *certificate authority*, of certifying the authenticity of a public key, i.e. that a public key is associated with a certain entity. Consequently, entities are not required to trust one another but need to establish a trust relationship with one or more certificate authorities. The authenticity of a public key is certified by appending a signature by a certificate authority. The result is called a *certificate*. Contrary to plain public key cryptography which requires entities to exchange public keys, this is not necessary with certificates because the authenticity of a certificate is established by verifying its signature against the signing certificate authority.

**Note:** By reducing the number of trust relationships in a public key infrastructure, entities are provided with a higher degree of security because there are less points of failure but the effect of a compromised certificate authority are more severe. This is one of the issues which are discussed in section 6.16.2 about risks and issues.

Due to its underlying cryptography, a Public Key Infrastructure is able to provide the following properties for the exchange of messages:

**Confidentiality.** By implementing public key cryptography, secure communication is enabled. Messages are encrypted with the public key of the recipient that only an entity in possession of the corresponding private key is able to read it them. Usually, the encryption of bulk data is performed by symmetric algorithms after the shared secret has been exchanged using public key cryptography. This allows for a much higher throughput.

**Authenticity.** By digitally signing a message, the recipient is assured that only an entity in possession of the sender's private key was able to create the message with the corresponding signature. Therefore, entities are able to authenticate against each other.

**Integrity.** A by-product of digital signatures is an inherent integrity check because only an entity in possession of the sender's private key was able to create the signature. Therefore, a modification results in an invalid signature.

**Non-repudiation.** Another by-product of digital signatures is that a sender cannot deny that his or her private key was used to sign a message.

## 5.1 Certificates

The *certificate* is a basic concept in Public Key Infrastructures because, in conjunction with the private key, it is used to authenticate entities, encrypt data, and sign messages.

It consists of a public key, the entity's identification information (the *subject*) and meta information, for example the validity. These three pieces of information are bound together by the signature of another entity (the *issuer*). This signature confirms that the public key belongs to the entity identified by the accompanying identity information (see Illustration 5). If any part of the signed data (public key, identity information, meta information) or the signature itself is modified, the certificate ceases to be valid because the signature was created based on different information.

**Warning:** In a certificate, an entity's identity information is explicitly bound to a public key but an entity is only implicitly associated with the public key.

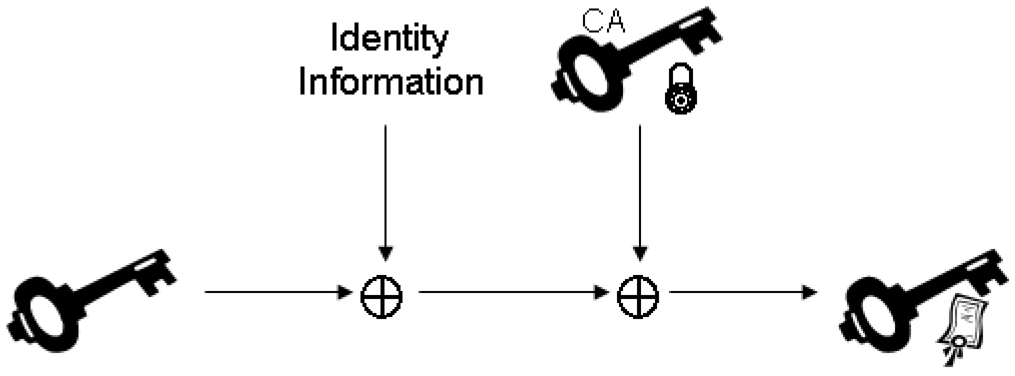

*Illustration 5: Certificate*

## 5.2 Certificate Authorities

Entities may acquire a special role, the *Certificate Authority (CA)* or *Certification Authority*. Its purpose is to accept an applicant's public key and identity information, successfully verify its validity and issue a certificate by binding the provided information using a digital signature (see Illustration 5). A major difference between implementations of Public Key Infrastructures lies in the number of entities required or allowed to take on the responsibilities of a Certificate Authority. A classification is presented in the next section.

The authority of a CA is not created implicitly by its existence but needs to be accepted explicitly by each entity. This is performed by adding the CA's certificate to the private collection of trusted certificates. The act of accepting a CA's authority is often performed by a system vendor instead of the end user so that the vendor needs to be trusted to assemble a list of trustful CAs.

By trusting a Certificate Authority, an entity assumes that the CA can be relied upon to perform its duty correctly. Consequently, signatures which verify against such a CA are trusted unconditionally.

## 5.3 Trust Models

When an entity decides to trust a Certificate Authority, this trust is expressed differently in implementations of Public Key Infrastructures with individual advantages and disadvantages (see Illustration 6).

**Direct.** The direct trust model does not support entities in creating trust relationships so that public keys need to be exchanged and managed manually for every communication endpoint. This is equivalent to implementing public key cryptography without any supporting infrastructure and results in a high overhead.

**Hierarchical.** In a hierarchical trust model, a small number of entities are assigned a special role and are acting as authorities. These authorities certify the ownership of public keys and form a tree in which authority is delegated to subordinate authorities. Only a small overhead is caused by the hierarchical trust model because only a small number of authorities is available. A X.509 Public Key Infrastructure, which is presented in section 6, is based on the hierarchical trust model.

**Distributed.** In a distributed trust model, all entities are considered to be equal. Consequently, every entity represents an authority and creates trust relationships with other entities. This model is different from the direct trust model because every entity may decide who is considered to be an authority as well trusting new keys automatically. A medium overhead is caused by the distributed trust model because every entity must decide which other entities to trust and to what extent. An OpenPGP Public Key Infrastructure, which is presented in section 7, is based on the distributed trust model.

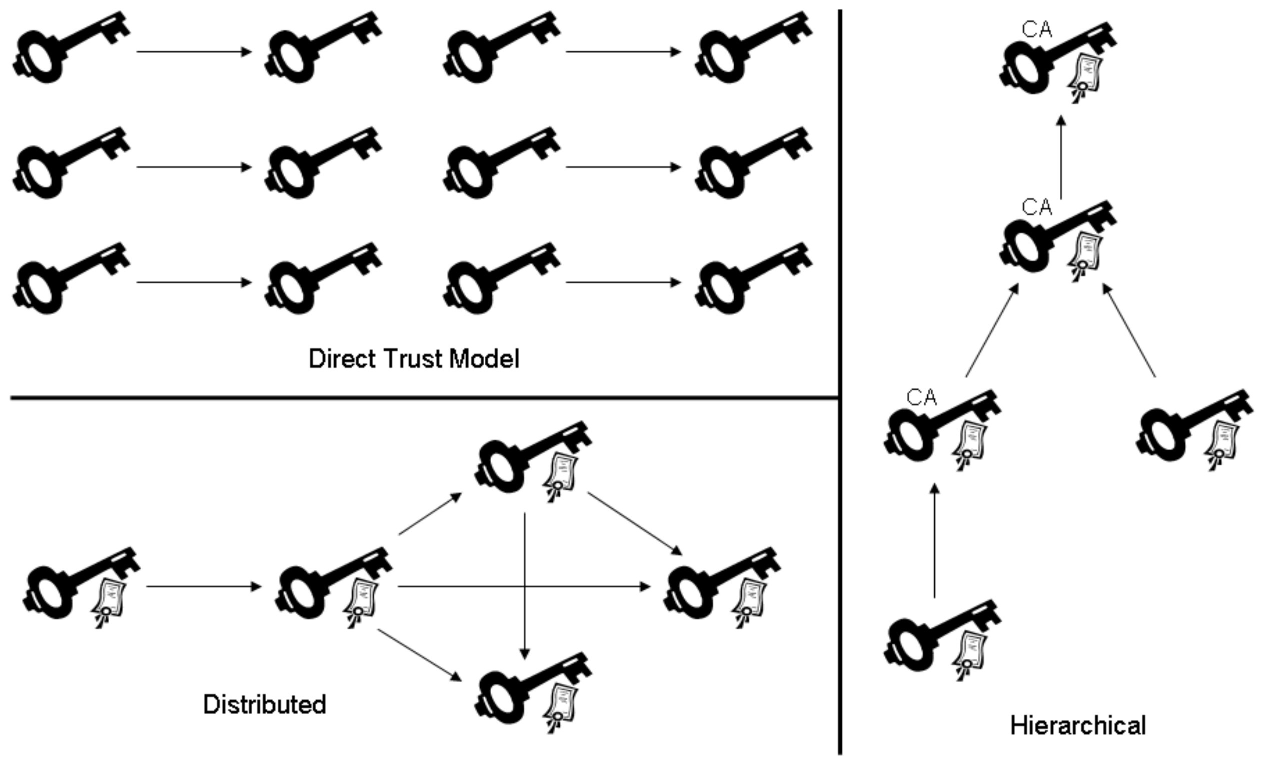

*Illustration 6: Trust Models*

## 5.4 Certificate Chains

If the validity of a certificate or signature needs to be verified, a certificate chain is created. This chain is built by a series of transitive relationships. It starts with the certificate in question and ends with a root certificate. For all intermediate certificates the subject must be matched with the issuer of the previous certificate (see Illustration 7). Before the certificate is considered to be valid, the root certificate needs to be trusted and all intermediate certificates in the chain need to be validated.

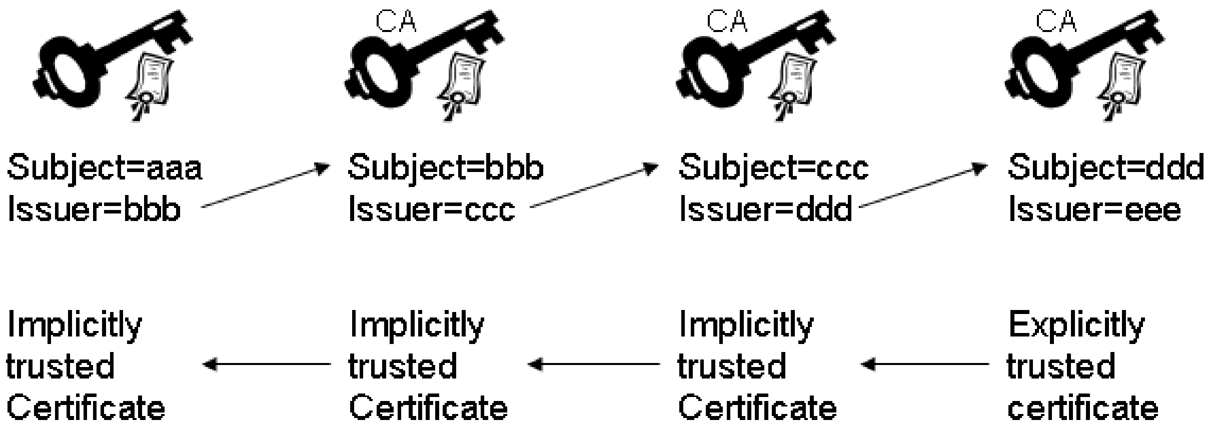
CA
CA
CA
Subject=aaa
Issuer=bbb
Subject=bbb
Issuer=ccc
Subject=ccc
Issuer=ddd
Subject=ddd
Issuer=eee
Implicitly trusted Certificate
Implicitly trusted Certificate
Implicitly trusted Certificate
Explicitly trusted certificate

*Illustration 7: Verification of a Certificate Chain*

## 5.5 Verification Models

After a certificate chain is created, all involved signature are checked against the signer's certificate. Although this might seem to be a trivial process, there are several models how to perform the verification and how to deal with expired certificates.

### 5.5.1 Classical Verification Model

This model ensures that all involved certificates are valid at the time the verification is performed (see ). Although this is the most intuitive approach, it also introduces a severe restriction to the lifetime of a signature because the validity of its signature depends on the validity of all certificate in the corresponding certificate chain.

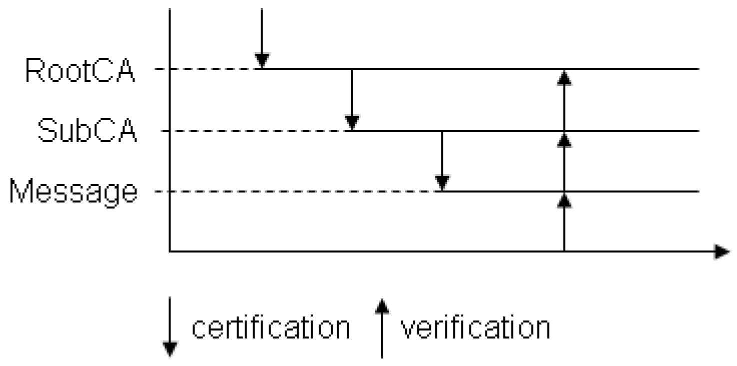

*Illustration 8: Classical Verification Model*

### 5.5.2 Modified Verification Model

Due to the shortcoming of the classical verification model, it was modified to require the verification to be backdated to the time of the creation of the message's signature (see Illustration 9). This model requires all certificates to be valid at the time of the creation of the signature in question. It also allows for one or more certificates in the chain to expire after the creation of the message's signature without causing the signature to expire.

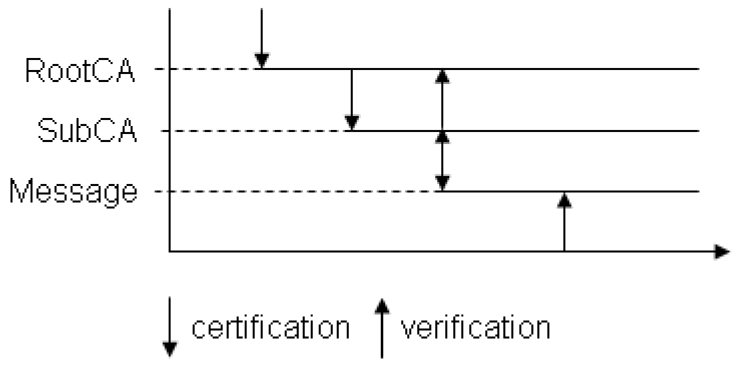

*Illustration 9: Modified Verification Model*

### 5.5.3 Backdating Verification Model

The backdating verification model allows for a much less restrictive connection between the validity of certificates and its signatures. It requires the message's signature and each certificate to be valid at the time each of them was created (see Illustration 10). Consequently, the verification of each item is backdated to its creation. Obviously, this model allows certificates to be used even after the certificate of the signing subordinate or even of the root Certificate Authority has expired.

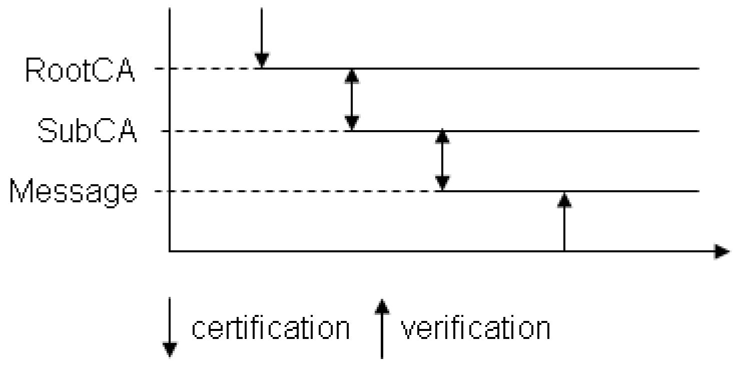

*Illustration 10: Backdating Verification Model*

## 5.6 *Enrolment, Renewal, Revocation*

In a Public Key Infrastructure, there are several tasks that need to be performed during the lifetime of a certificate. When a Certificate Authority signs a public key and the associated identity and, thereby, issues a certificate, the process is called enrolment. Due to the fact that a certificate contains a lifetime, it needs to be renewed (renewal process) before it expires. If the security of a private key is compromised, the corresponding certificate must be revoked (revocation process) and this information be published.

# 6 X.509 Public Key Infrastructures

*X.509* is a standard for Public Key Infrastructures by the *International Telecommunication Unit, Telecommunication Standardization Sector (ITU-T)*[27]. This standard was adopted as *X.509 Public Key Infrastructures*[28] by the *Public Key Infrastructure X.509 (PKIX)*[29] working group of the *Internet Engineering Task Force (IETF)*[30].

It implements a hierarchical trust model as explained in section 5.3. The overhead for this type of PKI is very low because only Certificate Authorities are required to certify the identity of entities. Consequently, entities only need to build a trust relationship to a collection of authorities to participate in the PKI.

A X.509 PKI does not support the distributed trust model because certificates can only contain a single signature. If a distributed X.509 PKI was built, the overhead for managing trust relationships would be proportional to the number of communication endpoints[31].

## 6.1 X.500 Directories

A *directory* is a specialized database that has been optimized for reading, browsing and searching. It is assumed that updates are not performed often and that individual updates are small in size. Directories are commonly based on the X.500 standards[32] and are accessed using the *Lightweight Directory Access Protocol (LDAP)*[33].

The data contained in a directory is organized in a hierarchical, tree-like structure which usually reflects geographic and/or organizational boundaries. An *entry* has a *Distinguished Name (DN)* which represents its position in the tree by a unique path from the root entry. It also contains a list of *attributes* each of which consists of a name, a type and a value. Attributes are used to store information about an entry. The part of a Distinguished Name which identifies an entry relative to its parent is called the *Relative Distinguished Name (RDN)*.

**Example:** The DN `/CN=Test/C=DE/ST=NRW/L=Cologne/O=Test GmbH/OU=IT` (see Illustration 11) represents an entry with the RDN `CN=Test`. The tree is build according to geographic and organizational boundaries because the DN contains elements specifying a country (C), a state or province (ST) and a locality (L) as well as elements specifying an organization (O) and an organizational unit (OU).

**Example:** The DN `/CN=Test/OU=Users/DC=Test/DC=Com` (see Illustration 11) represents an entry with the RDN `CN=Test`. The entries in the directory are solely

27 http://www.itu.int/ITU-T/
28 http://www.ietf.org/rfc/rfc3280.txt
29 http://www.ietf.org/html.charters/pkix-charter.html
30 http://www.ietf.org
31 Due to the fact that a X.509 certificate can only contain a single signature, an entity's public key would have to be bound in a certificate for every possible endpoint in order for the endpoint to be able to verify its authenticity. Apparently, this is not feasible.
32 X.500 defines a series of standards for directory services by the ITU-T.
33 http://www.ietf.org/rfc/rfc1777.txt and later RFCs

organized by organizational boundaries. The DN contains elements specifying several domain components (DC) and an organizational unit (OU).

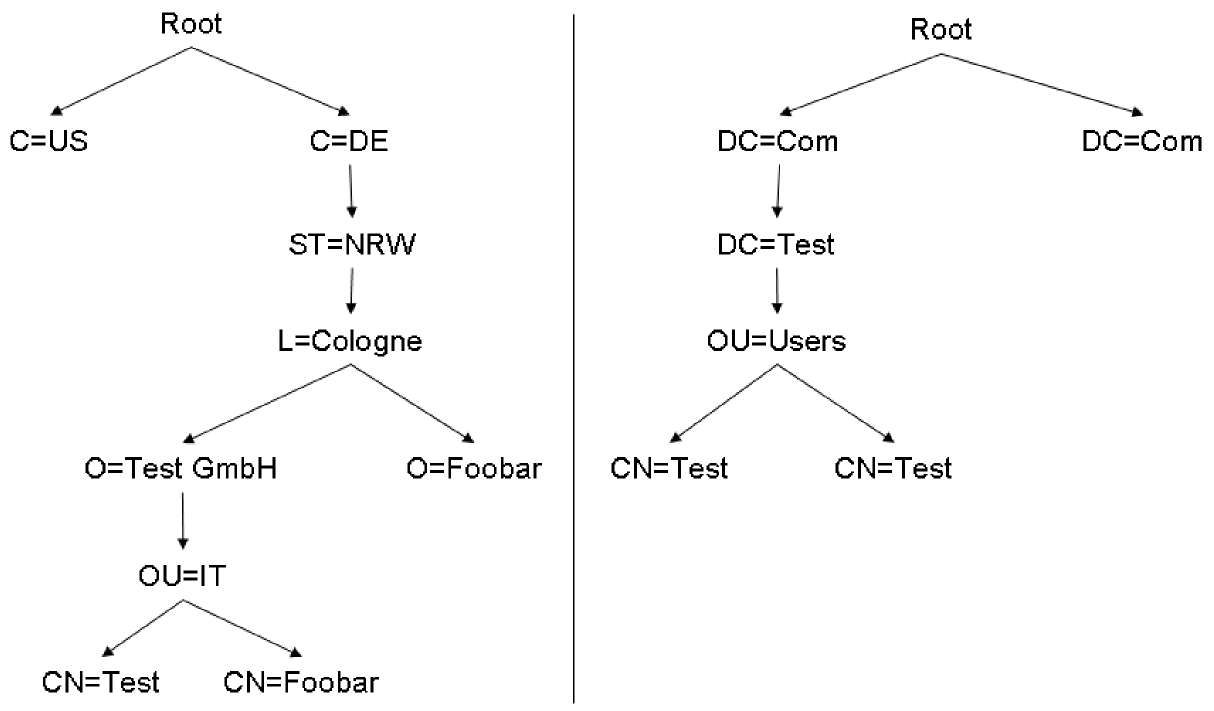

*Illustration 11: Directories*

X.500 based directories were originally intended to store a global directory of certificates to support lookup and retrieval. After the development of X.509 Public Key Infrastructures and the standardization by the IETF, X.500 is only used to represent an entity's identity information as a Distinguished Name.

### 6.1.1 Well-known Fields

In X.509 Public Key Infrastructures, a Distinguished Name consists of geographical as well as organizational RDNs. These are summarized in the following table:

| ***Field*** | ***Short name*** | ***Long name*** |
|---|---|---|
| Common Name | CN | commonName |
| Country | C | countryName |
| State | ST | stateOrProvinceName |
| Locality | L | localityName |
| Organization | O | organizationName |
| Organizational Unit | OU | organizationalUnitName |
| Email Address | - | emailAddress |

*Table 3: Well-known Fields in Distinguished Names*

## 6.2 X.509 Certificates

X.509 Certificates are based on the Abstract Syntax Notation 1 like private and public keys in public key cryptography (refer to section 4.4). Both are expressed either in the DER or the Base64 encoded DER format. By default, the OpenSSL library uses the latter.

In such a certificate, the Certificate Authority's signature binds a public key to an identity which is specified by a X.500 Distinguished Name or Alternative Name (email address or DNS name). It also contains several fields which contain important meta information about the certificate (refer to the example in section 6.2.2):

**Version.** This field specifies the version of the X.509 standard to be used. Due to the age of the X.509 standard and the availability of the third revision, this field can be expected to contain 3.

**Serial Number.** All certificates contain a serial number which is an iterator maintained locally by every Certificate Authority.

**Issuer.** This field contains the issuer's identity information represented as a X.500 Distinguished Name.

**Validity.** When a certificate is signed by the issuer, the validity is included requiring the subject to re-certify regularly with the Certificate Authority, because the certificate ceases to be valid and its verification fails per definition.

**Subject.** This field contains the subject's identity information represented as a X.500 Distinguished Name.

**Subject Public Key.** The public key contained in the certificate is bound to the subject's identity information.

**Issuer Signature.** The certificate is signed by the Certificate Authority's private key and, therefore, secures the stored information against modifications. Certificates can only contain a single signature due to the hierarchical nature of a X.509 Public Key Infrastructure.

### 6.2.1 Extensions

As of version 3, a X.509 certificate may contain a list of extensions which add further descriptions to the key. These extensions can be found in the human-readable representation in section 6.2.2.

**X509v3 Key Usage.** This field contains an abstract representation of the purpose of the certificate and specifies how the public key in the certificate may be used. The following key usages are defined: Digital Signatures, Non-repudiation, Key Encipherment, Data Encipherment, Key Agreement, Key Certificate Signing, CRL Signing.

**Netscape Certificate Type.** This field contains an representation of the purpose of the certificate and specified how the certificate itself may be used. The following certificate types are defined: SSL Server, SSL Client, S/MIME Client, Object Signing, SSL Server CA, SSL Client CA, S/MIME Client CA, Object Signing CA.

**X509v3 Basic Constraints.** Due to the fact that only some entities are considered to be Certificate Authorities, this field specifies whether the certificate may be used to issue new certificates.

**X509v3 Subject Alternative Name.** A subject may be known by an alternative name which is specified in this field. This can be an email address or a DNS name. Refer to section 6.15.8 for details.

Microsoft® Windows™ based systems honour these fields and require them to be present.

### 6.2.2 Example

The following box shows a X.509 certificate for reference:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            8f:6b:61:72:86:32:90:c5
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: CN=localhost
        Validity
            Not Before: Dec 27 17:13:11 2005 GMT
            Not After : Jan 26 17:13:11 2006 GMT
        Subject: CN=localhost
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:...:00
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            Netscape Cert Type:
                SSL Client, SSL Server, S/MIME
            X509v3 Subject Key Identifier:
                01:...:00
            X509v3 Authority Key Identifier:
                keyid:01:...:00
                DirName:/CN=localhost
                serial:8F:6B:61:72:86:32:90:C5

            X509v3 Key Usage:
                Digital Signature, Non Repudiation, Key Encipherment,
     Data Encipherment
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Subject Alternative Name:
                DNS:localhost
    Signature Algorithm: sha1WithRSAEncryption
        00:...:00
```

## 6.3 OpenSSL Configuration File

In comparison to the previous chapters, the complexity of the examples contained herein rises significantly requiring a detailed configuration file for the OpenSSL library. It contains several sections for the configuration of the aforementioned authority. The following example for a configuration file is taken from the SimpleCert project which features a free and stand-alone certificate store and Certificate Authority. The individual sections are discussed later in this chapter.

Please store the following data in a file named `openssl.config`:

```
[ca]
default_ca      = certificate_authority

[certificate_authority]
dir             = .
certs           = $dir/crt
new_certs_dir   = $dir/crt
crl_dir         = $dir/crl
database        = $dir/index
certificate     = $dir/ca_crt.pem
serial          = $dir/serial
crlnumber       = $dir/crlnumber
crl             = $dir/ca_crl.pem
private_key     = $dir/key/ca_key.pem
RANDFILE        = $dir/key/.rand
name_opt        = ca_default
cert_opt        = ca_default
default_crl_days= 30
default_days    = 365
default_md      = sha1
x509_extensions = ext_user
preserve        = no
policy          = policy_match

[policy_match]
commonName              = supplied
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
```

```
organizationName        = optional
organizationalUnitName  = optional
emailAddress            = optional

[policy_anything]
commonName              = supplied
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
emailAddress            = optional

[req]
default_bits            = 2048
default_keyfile         = ./key/ca_key.pem
default_md              = sha1
distinguished_name      = req_distinguished_name
attributes              = req_attributes

[req_distinguished_name]
commonName                        = Common Name
commonName_default                = example CA
countryName                       = Country
countryName_default               = DE
#countryName_min                   = 2
#countryName_max                   = 2
stateOrProvinceName               = State or province
stateOrProvinceName_default       = NRW
localityName                      = Locality
localityName_default              = Cologne
organizationName                  = Organization
organizationName_default          = example GmbH
organizationalUnitName            = Organizational Unit
organizationalUnitName_default = IT
emailAddress                      = Email address
emailAddress_default              = certmaster@example.com

[extensions_root_CA]
basicConstraints      = critical, CA:true
subjectKeyIdentifier  = hash
authorityKeyIdentifier=keyid:always,issuer:always
nsCertType            = objCA,emailCA,sslCA
```

```
subjectAltName        = email:copy
#nsCaRevocationUrl     = http://www.example.com/root_ca/ca_crl.pem
#crlDistributionPoints = URI:http://www.example.com/root_ca/ca_crl.pem
#nsCaPolicyUrl         = "http://www.example.com/root_ca/policy/"

[extensions_subordinate_CA]
basicConstraints      = critical, CA:true
subjectKeyIdentifier  = hash
authorityKeyIdentifier=keyid:always,issuer:always
nsCertType            = objCA,emailCA,sslCA
subjectAltName        = email:copy
#nsCaRevocationUrl     = http://www.example.com/root_ca/ca_crl.pem
#crlDistributionPoints = URI:http://www.example.com/root_ca/ca_crl.pem
#nsCaPolicyUrl         = "http://www.example.com/root_ca/policy/"

[extensions_user]
nsCertType             = server,client,email
#crlDistributionPoints  = URI:http://www.example.com/crl.shtml
#nsCaPolicyUrl          = http://www.example.com/policy.html
subjectAltName         = email:copy
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid,issuer:always
keyUsage               = digitalSignature, nonRepudiation,
     keyEncipherment, dataEncipherment
basicConstraints       = CA:false
nsComment              = "OpenSSL Generated Certificate"
```

## 6.4 Self-signed Certificates

Instead of having an authority sign a certificate, it can sign itself. In that case, you certify yourself that the contained information is correct (see Illustration 12). Although this does not provide any kind of authentication because you or an attacker can change the contents of the certificate at will, it allows for confidential communication and ensures message integrity.

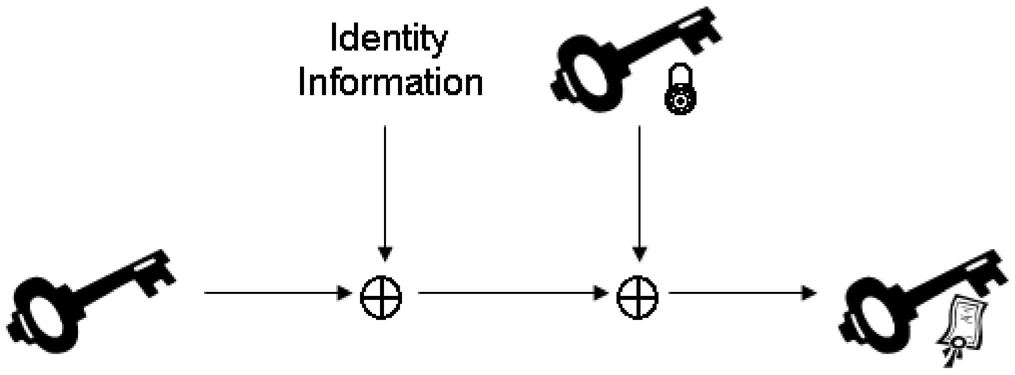

*Illustration 12: Self-signed Certificates*

Be aware that such *self-signed certificates* are nearly as critical as shared secrets because an Evil Person™ is able to create a new self-signed certificate which consists of the same information as the original certificate. It does not provide any kind of tamper-proof authentication because it is impossible to validate the claim of another entity to have a certain identity from the information contained in a self-signed certificate. Self-signed certificates are usually used for testing purposes only, although we will see that they are, in fact, the basis for every PKI.

The OpenSSL library allows you to create a self-signed certificate with a single command. As you will see the following commands also specify the OpenSSL configuration file (`-config`) and X.509 extensions (`-extensions`) that are added to the certificate. The following commands create a self-signed certificate based on a RSA and DSA private key file, respectively, the creation of which was explained in section 4.5.1.

- Create a 1024 bit RSA private key and a self-signed certificate[34]:

```
openssl req \
     -config openssl.config -extensions extensions_user \
     -x509 \
     -newkey rsa:1024 \
     -keyout self_rsa_key.pem \
     -out self_rsa_crt.pem
```

- Create a DSA private key and a self-signed certificate based on a DSA parameter file:

```
openssl req \
     -config openssl.config -extensions extensions_user \
     -x509 \
     -newkey dsa:host_dsa_prm.pem \
     -keyout self_dsa_key.pem \
     -out self_dsa_crt.pem
```

**Note:** The creation of a DSA parameter file was demonstrated in section 4.5.1.

34 http://www.openssl.org/docs/apps/req.html

## 6.5 Certificate Signing Requests

Although the previous section demonstrated how to create a self-signed certificate, it makes more sense to have a Certificate Authority sign your public key and, thereby, confirm that the contained information was correct at the time the signature was created. In order for someone else to sign your certificate, you need to create a *Certificate Signing Request (CSR)*[35].

A Certificate Signing Request always contains the subject's public key, the subject's identity information represented as a X.500 Distinguished Name, and is self-signed by the subject (see Illustration 13). Optionally, the CSR may contain a list of extensions which are expected to be included in the certificate.

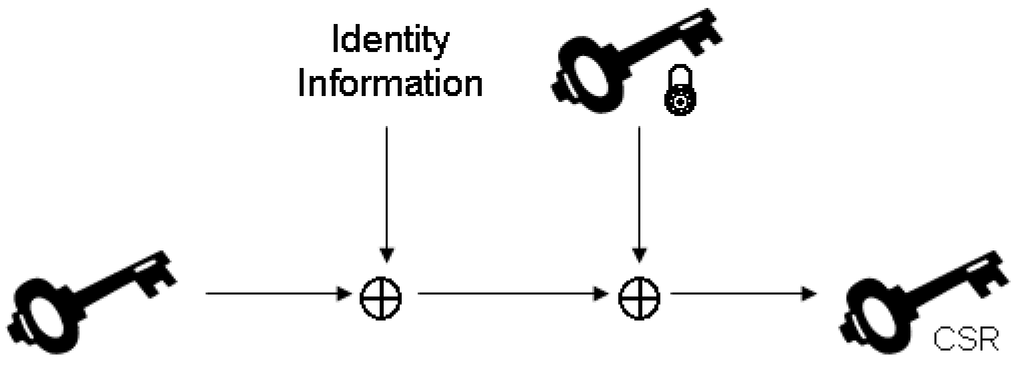

*Illustration 13: Certificate Signing Request*

The following excerpt contains a Certificate Signing Request in human-readable form. Obviously, the subject requests that an alternative name is included in the certificate:

```
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: CN=Test
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:...:00
                Exponent: 65537 (0x10001)
        Attributes:
        Requested Extensions:
            X509v3 Subject Alternative Name:
                DNS:Test
    Signature Algorithm: sha1WithRSAEncryption
        00:...:00
```

35 Certificate Signing Requests are defined in the Public Key Cryptographic Standard #10 (PKCS#10). See http://www.rsasecurity.com/rsalabs/node.asp?id=2124 for details.

### 6.5.1 Creating a CSR

When creating a CSR, the OpenSSL library asks you for the identity information that is required to sign the public key. The following list presents several flavours of creating a CSR with various prerequisites:

- Generate a CSR based on an existing RSA private key or key pair:

```
openssl req \
     -new \
     -key host_rsa_key.pem \
     -out host_rsa_csr.pem
```

- Generate a RSA private key and a CSR:

```
openssl req \
     -newkey rsa:1024 \
     -keyout host_rsa_key.pem \
     -out host_rsa_csr.pem
```

- Generate a CSR based on an existing DSA private key:

```
openssl req \
     -new \
     -key host_dsa_key.pem \
     -out host_dsa_csr.pem
```

- Generate a DSA private key and a CSR based on a DSA parameter file:

```
openssl req \
     -new \
     -newkey dsa:host_dsa_prm.pem \
     -keyout host_dsa_key.pem \
     -out host_dsa_csr.pem
```

Instead of entering the identity information on the command line, which can become rather cumbersome while familiarizing with the concepts and command line interface, the `-subj` parameter can be supplied to specify the identity information represented as a LDAP Distinguished Name. The following command creates a RSA private key and a CSR for a X.500 Distinguished Name containing the Common Name “Test”:

```
openssl req \
     -new \
     -newkey rsa:1024 \
     -subj "/CN=Test" \
     -outkey host_rsa_key.pem \
     -out host_rsa_csr.pem
```

The parameters required for signing are specified in the `[policy_match]` and `[policy_anything]` sections of the OpenSSL configuration file.

### 6.5.2 Viewing a CSR

The information contained in a Certificate Signing Request can be extracted and displayed in human-readable form. The following command applies to CSRs generated from RSA or DSA based private keys as explained in the previous section:

```
openssl req \
     -text -noout \
     -in host_rsa_csr.pem
```

**Note:** The `req` command also accepts parameters to access some fields of the Certificate Signing Request individually: `-pubkey`, `-modulus` and `-subject`.

### 6.5.3 Verifying a CSR

The signature of a Certificate Signing Request can be verified with the applicant's public key that was used to create it.

```
openssl req \
     -verify \
     -noout \
     -key host_rsa_key.pem \
     -in host_rsa_csr.pem
```

**Note:** Due to the fact that the public key is contained in the certificate, this procedure only ensures the integrity of the CSR and does not authenticate the applicant.

## *6.6 Certificate Authorities*

In a X.509 Public Key Infrastructure, the C*ertificate Authority (CA)* provides a central service. It certifies that a public key and the identity information of an entity belong together and binds them inside a certificate. To be reasonably sure of this claim, a CA defines all processes in its *Certificate Practices Statement (CPS)*. When placing trust in a CA, you usually decide that the CPS is acceptable for your needs and that you have reason to believe that the CA honours and enforces its own practices defined in this statement. In addition to creating certificates, a CA provides services for managing certificates during and after their lifetime, including enrolment, renewal and revocation.

Certificate Authorities come in two flavours:

**Root CAs.** A *Root CA* represents the top-level authority in a PKI. It must be trusted explicitly and unconditionally because it certifies its identity by distributing a self-signed certificate.

**Subordinate CAs.** A *Subordinate CA* is certified by another CA of any type to be able to provide the same services as the certifying CA. Its fitness for operation depends on its predecessor in the same way a certificate depends on a CA.

If a CA provides its services to the public, it needs to distribute its certificate to a wide range of devices in order to ensure that an issued certificate can be verified easily[36]. Note that "easily" does not imply "securely". Modern software usually incorporates a set of trusted public CAs. The user tends to blindly trust these certificates because the download and verification of a new certificate from a CA is too cumbersome.

### 6.6.1 Preparation

The first step in setting up a Certificate Authority is the creation of several directories and the initialization of its database. In the OpenSSL configuration file, the locations of these are specified in the first and second sections (`[ca]` and `[certificate_authority]`). In the `[ca]` section the `default_ca` parameter refers to the default configuration of a CA which is defined in the `[certificate_authority]` section. The configuration file, called `openssl.config`, is expected to reside in the same directory in which the following commands are executed.

The recommended directories for a CA are used to store the private key file, issued certificates and certificate revocations:

```
md \
     crt crl key
```

The initialization of the database is performed by the following commands which create two essential files. The index file contains the database which stores all issued and revoked certificates and the serial file which specifies the serial number of the certificate which will be issued next.

```
Type nul>index
Echo 01>serial
```

The contents of this directory make up a Certificate Authority consisting of the OpenSSL configuration file, the three directories and the two database files.

**Note:** In addition to the database file `index`, the `index.attr` file contains settings customizing the behaviour of the Certificate Authority. By settings the `unique_subject` parameter to „yes" instead of „no", the CA does not allow two valid certificates to contain the same subject DN. This is checked against the database.

For server certificates this parameter should be set to „no". During renewal, the old should be valid until the new certificate is installed. This allows for an easy migration as well as an easy rollback.

### 6.6.2 Creating a CSR for the CA

For both types of Certificate Authorities, root and subordinate, a Certificate Signing Request must be created. It contains the public key as well as the identity information of the authority.

36 For an in-depth discussion of public and private Root Certificate Authorities, please refer to sections 6.6.8 and 6.6.9.

Please refer to the section 6.4 for a demonstration of how CSRs are generated in various flavours. It is assumed that the CSR is saved to a file `ca_csr.pem` in the root directory of the CA.

### 6.6.3 Creating a Root CA

Based on a Certificate Signing Request, a Root Certificate Authority is created by generating a self-signed certificate[37]:

```
openssl ca \
     -config openssl.config \
     -name root_CA -extensions extensions_root_CA \
     -selfsign \
     -in ca_csr.pem \
     -out ca_crt.pem
```

In the case of a Root CA, these previous two steps (creating a CSR and the self-signed certificate) can be combined into a single command because no other authority is involved:

- Generate a self-signed certificate based on an existing RSA or DSA private key:

```
openssl req \
     -config openssl.config -extensions extensions_root_CA \
     -x509 \
     -new \
     -key ca_key.pem \
     -out ca_crt.pem
```

- Generate a RSA private key and a self-signed certificate:

```
openssl req \
     -config openssl.config -extensions extensions_root_CA \
     -x509 \
     -newkey rsa:1024 \
     -keyout ca_key.pem \
     -out ca_crt.pem
```

By default, every issued certificate is recorded in the CA's database file (`index`) of the CA. The first of the presented commands (implementing the `ca` command) causes the database to be updated accordingly. As a result, you need to take the two steps of creating a CSR and then self-signing it if an entry in the database is to be recorded.

**Note:** It is desirable to have a Root CA record all certificates including the one issued to itself as this allows for the tracking of all issued certificates.

The extensions parameter defines X.509 extensions that are added to the certificate (see the `[extensions_root_CA]` and `[extensions_subordinate_CA]` sections in the OpenSSL configuration file). For a CA, these parameters need to define that this is the certificate of a CA (among other things) and, therefore, allowed to sign certificates. Although OpenSSL nevertheless allows operations on certificates, the certificate of the CA

37 http://www.openssl.org/docs/apps/ca.html

may not be trusted by some certificate stores if it the signing CA did not confirm that it may be used for a Subordinate CA. Microsoft® Windows™ is known to rely on these X.509 extensions.

**Note:** There are some implications when choosing the validity of a root certificate. The effect of an expired root certificate is discussed in section 6.12.4.

### 6.6.4 Creating a Subordinate CA

To create a Subordinate Certificate Authority, a higher CA needs to sign the previously created CSR and, thereby, certify that it is allowed to issue certificates. In this step the higher CA needs to add the appropriate extensions which we were able to add by ourselves in the previous section. Again, some certificate stores may not allow the certificate of the Subordinate CA to be trusted if the appropriate extensions were not added by the signing CA.

The following command demonstrates how a higher CA signs a CSR generated by a new Subordinate CA:

```
openssl ca \
     -config openssl.config -extensions extensions_subordinate_CA \
     -in ca_csr.pem \
     -out ca_crt.pem
```

Although the extension sections for Root and Subordinate Certificate Authorities in the presented configuration file are very similar, separate sections are recommended because different settings will most likely be required in Real Life™.

**Note:** Please refer to section 6.2.1 for an introduction to X.509 extensions and section 6.6.6 for an explanation how they are incorporated in CSRs.

**Note:** There are some implications when choosing the validity of a Subordinate CA during enrolment. The effect of an expired certificate is discussed in section 6.12.4.

### 6.6.5 Enrolling a User Certificate

After the creation of a Subordinate Certificate Authority was demonstrated, the command used to *enrol* a user certificate is also based on a previously created CSR (refer to section 6.5.1 and Illustration 14). In fact, it only differs in the extensions added to the certificate:

```
openssl ca \
     -config openssl.cnf -extensions extensions_user \
     -in host_rsa_csr.pem \
     -out host_rsa_crt.pem
```

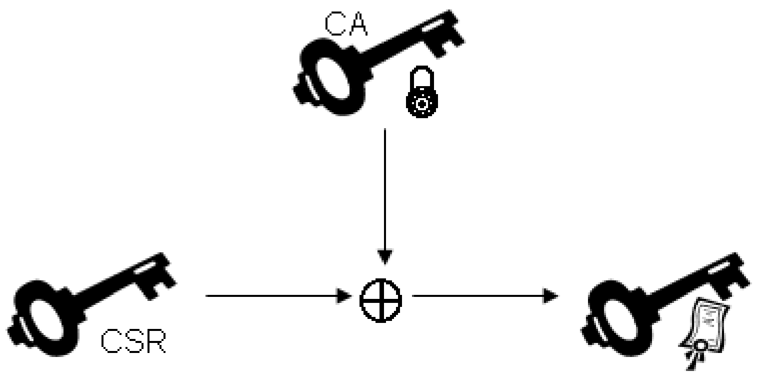

*Illustration 14: Generate a certificate from a Certificate Signing Request*

The resulting certificate can be viewed in human-readable form[38]:

```
openssl x509 \
     -text -noout \
     -in host_rsa_crt.pem
```

An important property of a certificate is the so-called fingerprint which reduces a certificate to a sequence of 40 hexadecimal digits (160 bits). It allows for an easy verification provided that the fingerprint for the certificate in question was obtained from a trusted location:

```
openssl x509 \
     -fingerprint -noout \
     -in host_rsa_crt.pem
```

**Further Study:** There are special parameters to output the individual fields of a certificate: `-serial`, `-subject_hash`, `-issuer_hash`, `-subject`, `-issuer`, `-email`, `-purpose`, `-dates`, `-modulus` and `-alias`.

### 6.6.6 Embedding X.509 Extensions

A Certificate Signing Request as well as a certificate may contain extensions specifying its usage to the Certificate Authority and the end user. In fact, Microsoft® Windows™ only recognizes certificates of a CA which has the appropriate extensions attached to it.

In this section, the presented commands demonstrate the creation of CSRs with the desired extensions. It is important that the CSR already contains the appropriate extensions because a CA does not necessarily know how the certificate is intended to be used.

- Create a CSR with extensions attached for the use as Root CA certificate:

```
openssl req \
     -config openssl.cnf -reqexts extensions_root_CA \
     -new \
```

38 http://www.openssl.org/docs/apps/x509.html

```
     -subj "/CN=Test" \
     -key host_rsa_key.pem \
     -out host_rsa_csr.pem
```

- Create a CSR with extensions attached for the use as Subordinate CA certificate:

```
openssl req \
     -config openssl.cnf -reqexts extensions_sub_CA \
     -new \
     -subj "/CN=Test" \
     -key host_rsa_key.pem \
     -out host_rsa_csr.pem
```

- Create a CSR with extensions attached for the use as an end user certificate:

```
openssl req \
     -config openssl.cnf -reqexts extensions_user \
     -new \
     -subj "/CN=Test" \
     -key host_rsa_key.pem \
     -out host_rsa_csr.pem
```

After a set of X.509 extensions has been embedded, it is at the CA's discretion to transfer these extensions into the resulting certificate. To achieve this, the configuration file needs to be modified to include the `copy_extensions` parameter in the CA section. Setting this parameter to `copy` only adds extensions that are not already present, whereas `copyall` enforces the extensions included in the CSR.

**Note:** The default configuration of OpenSSL presented in this document does not include the copy_extensions parameter because accepting the extensions included in a CSR must be considered to be a deliberate action because it bestows the owner of a certificate with additional rights.

**Note:** The attachment of extensions at the time of enrolment has already been demonstrated in the sections 6.6.3, 6.6.4, and 6.6.5.

### 6.6.7 Cross-signing a Certificate

Obviously, Certificate Authorities for a tree with a single Root CA at the top. This causes the design to be inflexible because the whole infrastructure depends on a single Root CA.

In real life, this restriction can be evaded by *cross-signing* between Root CAs which allows for a much larger number of entities due to the connection between two trees:

```
openssl ca \
     -config openssl.config \
     -preserveDN \
     -ss_cert ca2_crt.pem \
     -out ca2_crt2.pem
```

### 6.6.8 Utilizing a Public Root CA

When choosing a Public Root Certificate Authority, certificates will have to be purchased because these companies have a commercial interest in selling certificates. Depending on the number of certificates and the pricing model of such a public CA, it may not be viable to purchase them individually. At the same time, it is rather hard to get certified as a Subordinate CA because this effectively reduces the number of certificates that are purchased from the commercial CA. Although a CA may decide that such a delegation of authority is profitable because the applicant is committed for a long time because of the costly migration to another CA.

If you are required to provide a service to a large number of users external to your area of influence, the choice of a publicly verifiable certificate may be beneficial. The users can utilize any device that is able to access this service because a set of well-known root certificates is usually included in a modern system.

Well-known Root Certificate Authorities include Thawte, http://www.thawte.com, and Verisign, http://www.verisign.com. In addition, there is a non-commercial public Root CA called CAcert, http://www.cacert.org, which provides its service free of cost. Unfortunately, its root certificate has not yet been included in systems that are widely used. Petitions have been filed with the vendors.

### 6.6.9 Building a Private Root CA

By implementing your own independent Certificate Authority, you gain full control over its configuration in defining a sensible configuration and Certificate Practices Statement. This process also introduces the responsibility to maintain the CA and enforce the CPS. In addition, a number of resources need to be available for the successful operation:

- The hardware hosting the CA needs to be constantly available so that enrolment, revocation and validity verification are possible.
- The CA and the components which it is composed of need to be monitored, maintained and backed up to ensure its availability and security.
- To operate a CA, you need to supply the manpower to assume the aforementioned tasks as well as provide competent contacts.

If you are solely proving services for internal users who are exclusively using managed devices, a private Root CA can be easy to implement because the distribution of the root certificate can be achieved by an electronic software deployment tool. But such a PKI may become hard to replace if external access to the service is required at a later point in time.

There are some scenarios in which users will not be able to access a service that is secured by a private Root CA. Some proxy servers require an endpoint of a SSL/TLS connection to present a publicly signed certificate before the connection is accepted and data is forwarded to the user. These proxy servers may not always be under your influence and may severely hinder the access to your service.

You should also be concerned with the security of your Public Key Infrastructure. The service should not be implemented on a machine that is exposed on a network, physically as well as virtually. It is also common practice to isolate a CA in a virtual machine which can be backed up and stored in a safe. If the CA is utilized regularly, this may not be viable. Instead a Subordinate CA can be implemented to serve those requests. The Root CA can be safely stored away after certifying the Subordinate CA.

## 6.7 Trusted Certificates

The most important concept of Public Key Infrastructures is the trust between entities. But instead of manually establishing a trust relationship, entities simply need to trust a set of Certificate Authorities. Therefore, a collection of trusted certificates needs to be created in a dedicated directory, for example trust.

All *trusted certificates* are to be placed in this directory. Due to performance issues, the OpenSSL library does not verify certificates against this collection directly. Instead it accesses the trusted certificates by their hash. Therefore, we need to extract the hash of each certificate which we will call `HASH` from now on:

```
openssl x509 \
     -hash -noout \
     -in ca_crt.pem
```

If your operating system supports this, create a symbolic link pointing to the filename of the original certificate or otherwise rename the original file called `HASH.N` where `N` is a single-digit iterator in case certificates reduce to the same hash.

**Warning:** A certificate should not be easily considered trustworthy. To ensure that it was not compromised before or during the download, a unique attribute of the certificate must be verified. The fingerprint of a certificate is considered unique with a high probability. But both pieces of information, the certificate and the fingerprint, are to be obtained from separate and trusted sources to reduce the probability that both locations have been compromised simultaneously.

**Further Study:** Collections of trusted root certificates[39] are maintained by TeleTrust[40] and TACAR[41].

## 6.8 Certificate Chains

When complex trust relationships and structures of Certificate Authorities are used, the validity of a certificate can only be verified by a chain of certificates linking the certificate to a Root Certificate Authority. In addition, one or more intermediate CAs may be required to create a valid chain.

*Certificate chains* can be expressed in several formats. For the examples contained in this document, the PEM format will be used. Please refer to chapter 6.13 for conversions to

39 A set of root certificates can also be exported from the Microsoft Windows key store. Either export the certificates from Internet Explorer into a PKCS#7 structure and convert it to individual certificates (see section 6.14.2) or use `certutil.exe` from the Windows Server 2003 Admin Pak to export individual certificates.

40 http://www.teletrust.de/index.php?id=365

41 http://www.tacar.org/

different formats. A common format for certificates and certificate chains is PKCS#7 which is discussed in chapter 6.13.4.

### 6.8.1 Creating a Certificate Chain

A certificate chain in PEM format is trivially created by concatenating the involved certificates in a single file. All certificates are expected to be in PEM format.

### 6.8.2 Viewing Certificate Chains

If a certificate chain is contained in a PEM file, it cannot be viewed by the command presented in section 6.6.5. To view a certificate chain, convert it to PKCS#12[42] and then view its contents:

```
openssl pkcs12 \
     -export \
     -nokeys \
     -passout pass:”” \
     -in cert.pem \
     -certfile chain.pem \
| openssl pkcs12 \
     -passin pass:””
```

## *6.9 Certificate Tasks*

After the processes of creating Certificate Signing Requests and enrolling certificates have been covered, additional tasks are needed to manage and utilize a certificate during and after its lifetime.

### 6.9.1 Verifying a Signature

A certificate contains the signature of a Certificate Authority which confirms that the contained public key belongs to the entity indicated at by the enclosed identity information in the certificate. Using the certificate of the CA, this signature can be verified to ensure that the certificate is valid (see Illustration 15).

42 For details about this format, please refer to section 6.13.4.

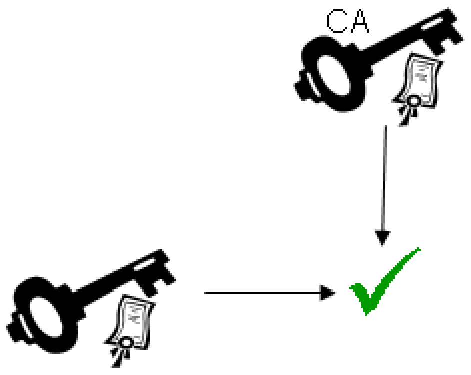

*Illustration 15: Certificate Verification*

If your installation of the OpenSSL library recognizes a list of trusted certificates by default, the following command verifies the validity of a certificate against this list[43]:

```
openssl verify \
     host_rsa_crt.pem
```

Otherwise you can either verify a certificate against a specific certificate of a trusted Certificate Authority:

```
openssl verify \
     -CAfile ca_crt.pem \
     host_rsa_crt.pem
```

The file `ca_crt.pem` may also contain a certificate chain.

Or have it verified against a directory containing trusted certificates as described in the previous section:

```
openssl verify \
     -CApath trust \
     host_rsa_crt.pem
```

In addition to the verification against trusted certificates, the verify command can also be used to verify that the certificate was created for a given purpose. This task can also be used in combination with the `-CAfile` and `-CApath` parameters.

```
openssl verify \
     -purpose PURPOSE
     host_rsa_crt.pem
```

The `-purpose` parameter expects one of the following values to be specified: `sslclient`, `sslserver`, `nssslserver`, `smimesign`, and `smimeencrypt`.

If a certificate verifies successfully, this is indicated by a message on standard output and the return code of the OpenSSL command line tool.

**Warning:** If the file contains more than one certificate, only the first is verified. The remaining certificates are not even used for verification. Therefore, a chain of certificates

43 http://www.openssl.org/docs/apps/verify.html

of intermediate authorities which are not explicitly trusted can be supplied by the -untrusted parameter.

### 6.9.2 Data Encryption

Analogous to public key cryptography, the certificate is used instead of the public key because, in fact, it is a public key with some accompanying information (see Illustration 3). While data is decrypted using the same commands presented in section 4.6, the command for the encryption changes slightly because a certificate is required instead of a public key:

```
openssl rsautl \
     -encrypt \
     -certin -inkey host_rsa_crt.pem \
     -in plaintext.txt \
     -out ciphertext.txt
```

### 6.9.3 Verifying Signed Data

Because of the fact that signatures are generated by encrypting data with the sender's private key, only the commands used for the verification of a signature changes to specify the certificate instead of the public key (see Illustration 4):

```
openssl rsautl \
     -verify \
     -certin -inkey host_rsa_crt.pem \
     -in ciphertext.txt \
     -out plaintext.txt
```

## *6.10 Certificate Revocation*

If a certificate has fulfilled its purpose or if the corresponding private key has been compromised or lost, it needs to be revoked. The certificate can no longer be considered valid and trustworthy.

### 6.10.1 Revoking a Certificate

The Certificate Authority needs to be informed of a compromised or lost certificate to *revoke* it as soon as possible:

```
openssl ca \
     -config openssl.cnf \
     -revoke host_rsa_crt.pem
```

When a certificate is revoked, the reason for this action can be specified by `-crl_reason` which takes one of the following parameters: `keyCompromise`, `CACompromise`, `affiliationChanged`, `superseded`, and `cessationOfOperation`.

If the private key was compromised resulting in the revocation of the corresponding certificate, it is helpful for other entities to learn the time this was detected. To embed this time in the revocation, the `-crl_compromise` and `-crl_ca_compromise` parameter are used. The revocation reason is automatically set to `keyCompromise` and `CACompromise`, respectively. Both parameters expect the time to be supplied in the generalized time format (`YYYYMMDDHHMMSSZ`) with `Z` representing the time zone.

### 6.10.2 Generating a Certificate Revocation List

After a certificate has been revoked, only the owner and the Certificate Authority are aware of the fact, that a certain certificate is no longer valid. The certificate of every CA needs to contain distribution points, e.g. a Uniform Resource Locator (URL), where a *Certificate Revocation List (CRL)* can be retrieved. The CRL contains a list of certificates that have been revoked and needs to be generated and published regularly to avoid the abuse of compromised private keys.

**Warning:** To gain knowledge of the certificates revoked by a trusted CA, a client needs to check the CRL distribution point before performing certificate tasks.

The following command generates a CRL:

```
openssl ca \
     -config openssl.cnf \
     -gencrl \
     -out ca_crl.pem
```

**Note:** On Microsoft® Windows™ platforms, certificates are exchanged using the DER format which requires the generated CRL to be converted from the PEM format. For details, please refer to section 6.13.

**Further Study:** A Certificate Revocation List contains a field which specifies when the next update is due. It is modified using the `-crldays` or `-crlhours` parameters and displayed using the `-nextupdate` parameter in section 6.10.5.

### 6.10.3 Publishing a CRL

After generation, a Certificate Revocation List needs to be published in order for users to be able to access, process and honour it. The location of the CRL can be embedded in issued certificates. To personalize the location, refer to the sections `[extensions_root_CA]` and `[extensions_subordinate_CA]` in the OpenSSL configuration file `openssl.config`. The following configuration options specify the well-known location where the CRL is published.

```
nsCaRevocationUrl      = http://www.example.com/root_ca/ca_crl.pem
crlDistributionPoints = URI:http://www.example.com/root_ca/ca_crl.pem
```

**Note:** This location is required to be online and accessible at all times. Once certificates are issued, the location may not be relocated or clients verifying the certificate would not able to check for a list of revoked certificates.

### 6.10.4 Verifying a CRL

Although the URL of the Certificate Revocation List is published in the certificate of a Certificate Authority, an entity should not blindly assume that this list is uncompromised. Therefore, a CRL is signed by the responsible CA and can be verified using the following command[44]:

```
openssl crl \
     -noout \
     -CAfile ca_crt.pem \
     -in ca_crl.pem
```

**Note:** In the case of a Subordinate CA, the signature needs to be verified against a collection of trusted certificates as outlined in section 6.9.1.

**Further Study:** The individual fields of a Certificate Revocation List are displayed using the `-hash`, `-fingerprint`, `-issuer`, `-lastupdate`, and `-nextupdate` parameters. The last parameter displays the field which is customized by the `-crldays` parameter in section 6.10.2.

### 6.10.5 Viewing a CRL

The list of revoked certificates can also be shown in human-readable form:

```
openssl crl \
     -text -noout \
     -in ca_crl.pem
```

**Further Study:** There are several more parameters which allow for the retrieval of individual fields from the Certificate Revocation List: `-hash`, `-fingerprint`, `-issuer`, and `-last`.

The following excerpt contains a Certificate Revocation List in human-readable form:

```
Certificate Revocation List (CRL):
        Version 1 (0x0)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: /CN=Test CA/C=DE/ST=NRW/L=Cologne/O=Test
     GmbH/OU=IT/emailAddress=certmaster@test.de
        Last Update: Dec 28 12:51:20 2005 GMT
        Next Update: Jan 27 12:51:20 2006 GMT
Revoked Certificates:
    Serial Number: 02
        Revocation Date: Dec 28 12:51:15 2005 GMT
    Signature Algorithm: sha1WithRSAEncryption
        00:...:00
```

44 http://www.openssl.org/docs/apps/crl.html

**Note:** The certificates are only listed by their serial number which does not allow the Distinguished Name of the revoked certificate to be derived without knowledge of the serial number because the details of the certificate can be misused to create a spoofing attack.

### 6.10.6 Status of a Certificate

If you are in control of the responsible Certificate Authority, you can also explicitly check the status of a certificate by its serial number `SERIAL`:

```
openssl ca \
     -config openssl.config \
     -status SERIAL
```

The serial number can be obtained from the header of a certificate. Refer to the human-readable representation of a certificate as outlined in section 6.2.2.

**Further Study:** Refer to section 6.12.2 how to check for expired certificates which also require a new Certificate Revocation List to be published.

## *6.11 Online Certificate Status Protocol*

An alternative to Certificate Revocation Lists is offered by the *Online Certificate Status Protocol (OCSP)*[45] which was developed by the Internet Engineering Task Force (IETF). It provides timely information about the status of a certificate by having the client device inquire directly instead of relying on a pre-compiled CRL. Unfortunately, this protocol is implemented by few browsers[46]. The CA is required to provide an additional service called the OCSP responder which is served via HTTP or HTTPS. Clients are able to send OCSP request to the responder and receive OCSP responses.

**Note:** Contrary to the default behaviour of the OpenSSL library, OCSP requests and responses are encoded using the DER format due to its network oriented nature. For details about conversions to and from the PEM format, please refer to section 6.13.

### 6.11.1 Requests and Responses

This section demonstrates the creation of OCSP requests and the retrieval of OCSP responses as well as the extraction of details from them in a human-readable representation[47]:

- Create an OCSP request for the certificate `cert.pem` which was issued by the Certificate Authority contained in `issuer.pem` and write it to the file `ocsp_req.der`:

  ```
  openssl ocsp \
       -issuer issuer.pem \
  ```

45 http://www.ietf.org/rfc/rfc2560.txt
46 The Open Source browser Mozilla Firefox supports the Online Certificate Status Protocol.
47 http://www.openssl.org/docs/apps/ocsp.html

```
     -cert cert.pem \
     -reqout ocsp_req.der
```

- View a request:

```
openssl ocsp \
     -text \
     -reqin ocsp_req.der
```

- Send the OCSP request to a responder at the URL http://localhost:8888 and write the response to the file `ocsp_resp.der`:

```
openssl ocsp \
     -url http://localhost:8888 \
     -reqin ocsp_req.der \
     -respout ocsp_resp.der
```

- Create and send the OCSP request to the responder and write the response to the file `ocsp_resp.der`:

```
openssl ocsp \
     -issuer issuer.pem \
     -cert cert.pem \
     -url http://localhost:8888 \
     -resp_text \
     -respout ocsp_resp.der
```

- View an OCSP response:

```
openssl ocsp \
     -text \
     resp.der
```

**Note:** An OCSP request may contain a list of certificates by specifying several `-cert` parameters.

**Note:** To secure the communication with the responder, user the `-signer` and `-signkey` parameters to sign OCSP requests. The verification of OCSP responses is performed using the `-CAfile` or the `-CApath` parameters.

### 6.11.2 Responder

The Online Certificate Status Protocol responder represents an additional service which a Certificate Authority needs to provide to entities. It accepts requests and returns responses containing the status of one or more requested certificates. The `respcert.pem` file contains a certificate and the corresponding private key which are used to sign responses. They may be separated by specifying the private key using the `-rkey` parameter.

```
openssl ocsp \
     -index index \
     -CA ca_crt.pem \
     -port 8888 \
     -rsigner respcert.pem \
     -text
```

**Note:** By adding the `-resp_no_certs` parameter, the response will not include any certificates.

### 6.11.3 Miscellaneous

The Online Certificate Status Protocol can also be used to verify certificates internally against the local database of the Certificate Authority:

- Verify a certificate identified by the serial number `SERIAL` internally:

```
openssl ocsp \
     -index index \
     -CA ca_crt.pem \
     -rsigner respcert.pem \
     -issuer ca_crt.pem \
     -serial SERIAL
```

- Verify a certificate internally using an OCSP request read from the file `ocsp_req.der` and write the response to the file `ocsp_resp.der`:

```
openssl ocsp \
     -index index \
     -CA ca_crt.pem \
     -rsigner respcert.pem \
     -reqin ocsp_req.der \
     -respout ocsp_resp.der
```

**Note:** The internal verification of a certificate requires the appropriate access permissions to the database file.

## *6.12 Certificate Expiry and Renewal*

During the creation of a certificate, a field indicating the lifetime is included in the certificate. This lifetime cannot be altered lest the certificate ceases to be valid. As soon as a certificate looses its validity, other entities will not accept signed and/or encrypted messages from its owner because the CA requires that the contained information is validated regularly.

Some Certificate Authorities notify their users when an issued certificate that is expected to expire in the near future. You should not rely on such a facility because there are several situations in which the notification may not reach you, e.g. the service is not operable (it failed unexpectedly or the network connection was interrupted either intentionally or not) or you have changed your email address.

### 6.12.1 Checking for Certificate Expiry

A certificate can be checked by an entity for expiry in the next `N` seconds using the following command:

```
openssl x509 –checkend N –in crt.pem
```

Whether the specified certificate expires in the set timeframe is expressed by the return code of the command and a message on standard output.

### 6.12.2 Checking for Expired Certificates

If you are in control of the responsible Certificate Authority, you are required to update the database of your CA on a regular basis to identify expired certificates:

```
openssl ca \
     -config openssl.config \
     -updatedb
```

You are also required to update your Certificate Revocation List after a database update so that it always contains the latest list of expired and revoked certificates (see section 6.10.2 for details).

### 6.12.3 Renewing a Certificate

A certificate can only be renewed if the corresponding Certificate Authority confirms that the public key still belongs to the identify information provided by the subject and creates a new certificate. It is not possible to modify the original certificate to contain a new lifetime because a new signature needs to be created.

This process is initiated by providing the CA with a Certificate Signing Request. One of the following variants produce an appropriate CSR:

- Send the same CSR that was used to create the original certificate.
- Create a new CSR as outlined in section 6.5.1.
- Create a new CSR from the original certificate (see Illustration 16):

```
openssl x509 \
     -x509toreq \
     -signkey host_rsa_key.pem \
     -in host_rsa_crt.pem \
     -out host_rsa_csr.pem
```

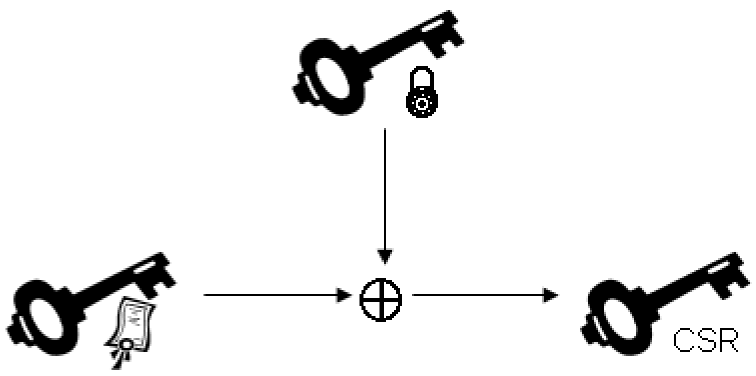

*Illustration 16: Generate a CSR from a certificate*

Although the CSR results in a new signature and, therefore, a new certificate, it still contains the same information: the same public key and the same identify information. Apart from the new lifetime, the certificate also reduces to a different fingerprint which you will have to update in case it was published.

**Note:** This process does not change the hash of the certificate which allows trusted certificates to be easily exchanged.

### 6.12.4 Renewing CA Certificates

For a Certificate Authority, the impact of a certificate expiry can be rather severe, because this may cause all issued certificates to loose their validity. No matter how well the renewal of a CA certificate was planned, the new certificate still needs to be distributed. For Root CAs which have been able to include their certificate in the release of an operating system, this may mean that they will not be able to publish the updated certificate to all clients.

The renewal process of CA certificates does not deviate from that for user certificates so that the information contained in the previous section applies. As long as the underlying private key is not changed, a new CSR may be generated and the certificate created without the issued certificates loosing their validity:

```
move ca_crt.pem ca_crt_orig.pem
openssl x509 \
     -x509toreq \
     -signkey key/ca_key.pem \
     -in ca_crt_orig.pem \
     -out ca_csr.pem
openssl x509 \
     -req \
     -signkey key/ca_key.pem \
     -in ca_csr.pem \
     -out ca_crt.pem
```

**Note:** The validity of the all previously issued certificates can be checked by verifying them against the new CA certificate.

**Warning:** If the private key of a CA was compromised or lost, it does not suffice to create a new private key and a corresponding certificate and distribute the certificate because clients are unable to verify certificates which were issued before the loss. In this case, the CA needs to create a new private key, obtain a new certificate and re-enrol all previously issued certificates. Consequently, this places a huge overhead on the CA as well as all its customers.

## *6.13 Conversions*

The default format utilized in the OpenSSL library for operations on key pairs, certificates and revocation lists is the *Privacy Enhanced Mail (PEM)* format which is Base64 encoded.

To allow the generated files to be used on other platforms or with other products, this chapter features conversions to and from different encodings and formats.

The formats presented in this chapter are all based on the Abstract Syntax Notation 1 (ASN.1) by the ISO and ITU-T. This standard defines several notations including Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) and provides the basis for X.500 directories.

### 6.13.1 Base64

*Base64*[48] is an encoding for binary data using 64 printable ASCII characters. It allows for such data exchange on a channel which reserves certain characters for connection, session or flow control:

- Data is converted into Base64 with the following command[49]:

```
openssl enc \
     -e \
     -base64 \
     -in file.ext \
     -out file.ext.base64
```

- The encoding is reversed with the following command:

```
openssl enc \
     -d \
     -base64 \
     -in file.ext.base64 \
     -out file.ext
```

**Note:** Do not confuse this with encryption because an encoding is performed and reversed without a shared secret and, therefore, does not provide any security against intentional attacks.

### 6.13.2 Privacy Enhanced Mail

All elements of public key cryptography and X.509 Public Key Infrastructures are processed in the Privacy Enhanced Mail (PEM) format which is a Base64 encoded DER format.

The encoded data is enclosed in start and end tags which also define the type of data:

- Private key:

```
-----BEGIN RSA PRIVATE KEY-----
MI...
-----END RSA PRIVATE KEY-----
```

- Public Key

48 http://www.ietf.org/rfc/rfc3548.txt
49 http://www.openssl.org/docs/apps/enc.html

```
-----BEGIN PUBLIC KEY-----
MI...
-----END PUBLIC KEY-----
```

- Certificate Signing Request

```
-----BEGIN CERTIFICATE REQUEST-----
MI...
-----END CERTIFICATE REQUEST-----
```

- Certificate

```
-----BEGIN CERTIFICATE-----
MI...
-----END CERTIFICATE-----
```

- Certificate Revocation List

```
-----BEGIN X509 CRL-----
MI...
-----END X509 CRL-----
```

### 6.13.3 Distinguished Encoding Rules

Although the OpenSSL library can be used to convert all PEM encoded files into *Distinguished Encoding Rules (DER)*[50] encoded files (.DER, .CER), Microsoft® Windows™ only recognizes DER encoded certificates without private keys. Therefore, the following command creates a valid DER encoded certificate which can be imported into a key store on a Microsoft® Windows™ platform:

```
openssl x509 \
     -in host_rsa_crt.pem \
     -outform DER -out host_rsa_crt.der
```

Nevertheless, the DER format can be utilized to perform all operations supported by the OpenSSL library. You can specify the input and output format of a key with the `-inform` and `-outform` parameters:

- Conversion of a DER formatted file into the PEM format:

```
openssl rsa \
     -in host.der \
     -outform PEM -out host.pem
```

- Conversion of a PEM formatted file into the DER format:

```
openssl rsa \
     -in host.pem \
     -outform DER -out host.der
```

50 DER is a method for encoding digitally signed data objects such as X.509 certificates.

**Further Study:** The conversion to and fro between PEM and DER can also be applied to certificates (`x509` command), Certificate Signing Requests (`req` command) and Certificate Revocation Lists (`crl` command) in an equivalent manner.

### 6.13.4 Public Key Cryptography Standard #12

On Microsoft® Windows™ platforms, only *Public Key Cryptography Standard #12 (PKCS#12)*[51] *a.k.a. Personal Information Exchange (PIX)* formatted files (.P12, .PFX) can be used to exchange certificates with the corresponding private keys[52]:

- Convert a certificate and the corresponding private key stored in separate files into a single PKCS#12 formatted file:

```
openssl pkcs12 \
     -export \
     -inkey host_rsa_key.pem \
     -in host_rsa_crt.pem \
     -out host_rsa_crt.p12
```

- Convert a certificate and the corresponding private key stored in a single file into a single PKCS#12 formatted file:

```
openssl pkcs12 \
     -export \
     -in host_rsa.pem \
     -out host_rsa.p12
```

- Convert a PKCS#12 formatted file into a single PEM formatted file:

```
openssl pkcs12 \
     -in host_rsa.p12 \
     -out host_rsa.pem
```

**Further Study:** The amount of data that is converted can be controlled using the following parameters: `-nocerts`, `-clcerts`, `-cacerts`, and `-nokeys`.

### 6.13.5 Public Key Cryptography Standard #7

The Public Key Cryptographic Standard #7 (PKCS#7)[53] defines a syntax for data with digital signatures or digital envelopes. A PKCS#7 structure may include plaintext and ciphertext, a signature, one or more certificates and Certificate Revocation Lists.

The `crl2pkcs7` command allows for Certificate Revocation Lists as well as certificates to be encoded in a PKCS#7 structure[54]:

51 PKCS#12 defines a format for exchanging private keys with the corresponding certificates and a symmetrical encryption.
52 http://www.openssl.org/docs/apps/pkcs12.html
53 http://www.rsasecurity.com/rsalabs/node.asp?id=2124
54 http://www.openssl.org/docs/apps/crl2pkcs7.html

```
openssl crl2pkcs7 \
     -in crl.pem \
     -certfile cert.pem
```

Although the name of the `-print_certs` parameter implies that only certificates are displayed, it splits a PKCS#7 structure into its individual PEM-encoded components:

```
openssl pkcs7 \
     -print_certs \
     -in input.p7c
```

Unfortunately, there is no parameter causing `pkcs7` to output all fields of the PKCS#7 structure.

## *6.14 S/MIME*

The *Secure MIME (S/MIME)*[55] standard is an extension to the *Multipurpose Internet Mail Extensions (MIME)*[56] which defines the format of an email message body and attachments sent over the internet. Both standards have been created by the PKIX working group of the IETF. S/MIME adds facilities to send encrypted and signed content in the message body as well as attachments with certificate chains. It is built on the *Cryptographic Message Standard (CMS)*[57] which is based on the *Public Key Cryptographic Standard #7 (PKCS#7)* and the Privacy Enhanced Mail (PEM) standard which is the default format for operations by the OpenSSL library.

### 6.14.1 Signing a Message

When signing a message, the OpenSSL library requires the private key as well as the certificate of the sender to be specified[58]:

- Sign a message by supplying a key pair:

```
openssl smime \
     -sign \
     -signer sender.pem \
     -nocerts \
     -in plaintext.txt \
     -out smimetext.txt
```

- Sign a message by supplying separate private and public keys:

```
openssl smime \
     -sign \
     -signer sender_crt.pem \
     -inkey sender_key.pem \
     -nocerts \
```

55 http://www.ietf.org/rfc/rfc2633.txt
56 Part One: http://www.ietf.org/rfc/rfc2045.txt, Part Two: http://www.ietf.org/rfc/rfc2046.txt, Part Three: http://www.ietf.org/rfc/rfc2047.txt
57 http://www.ietf.org/rfc/rfc3852.txt
58 http://www.openssl.org/docs/apps/smime.html

```
     -in plaintext.txt \
     -out smimetext.txt
```

In both examples, the sender's certificate is not embedded into the message. To do so, omit the `-nocerts` parameter:

```
openssl smime \
     -sign \
     -signer sender.pem \
     -in plaintext.txt \
     -out smimetext.txt
```

If the signing certificate can only be verified with certificates from one or more intermediate Certificate Authorities, they should be included in the S/MIME message:

```
openssl smime \
     -sign \
     -signer sender.pem \
     -certfile intermediate.pem \
     -in plaintext.txt \
     -out smimetext.txt
```

Note: If a protected private key is used, the passphrase can be specified using the `-passin` parameter (see section 4.5.2).

**Note:** It is not a security risk to embed a certificate because its signature can be verified by the appropriate certificate chain which needs to be present at the sender. Embedding a public key may result in a security breach because no authentication of the sending entity is provided and an attacker might be able to replace the whole message and embed a new public key. As a result, the recipient might be unable to detect the forgery, and accept the message.

### 6.14.2 Verifying a Signature

Similar to the verification of signed data with key pairs, an S/MIME encoded message can be verified by one of the following commands:

- Verify the message against the built-in trusted certificates:

```
openssl smime \
     -verify \
     -in smimetext.txt
```

- Verify the message explicitly against the signer's certificate:

```
openssl smime \
     -verify \
     -certfile host_rsa.pem \
     -in smimetext_signed.txt
```

- Verify the message against a collection of trusted certificates (refer to section 6.7 on how to create such a collection):

```
openssl smime \
     -verify \
     -CApath trusts \
     -in smimetext_signed.txt
```

- Verify the message against a specific certificate or certificate chain:

```
openssl smime \
     -verify \
     -CAfile ca_crt.pem \
     -in smimetext.txt
```

The previous section demonstrates how to embed a certificate into a S/MIME encoded message. The following commands show how such a certificate can be extracted:

- Extract and save an embedded certificate or certificate chain:

```
openssl smime \
     -pk7out \
     -in smimetext.txt \
     -out host_crt.pem
```

- Extract and display an embedded certificate or certificate chain[59]:

```
openssl smime \
     -pk7out \
     -in smimetext.txt \

| openssl pkcs7 \
     -print_certs
```

Note: For more details about the `pkcs7` command, refer to section 6.13.5.

The process of verifying a signature consists of two steps. The first ensures that the certificate contained in the S/MIME message is valid by verifying it against trusted certificates. The second step ensures that the signature in question verifies successfully against the certificate. Both steps can be suppressed individually by the `-noverify` and `-nosigs` parameters, respectively.

When the first step is performed, the OpenSSL library may encounter CA certificates which are not explicitly trusted by the recipient. Usually, these are automatically used for verification. This behaviour is inversed by the `-nochain` parameter resulting in an unsuccessful verification if untrusted CA certificates are to be used which are embedded in the message.

**Note:** Check the revocation status of the signer's certificate by adding the `-crl_check` parameter. To extend this behaviour to all certificates in the signer's certificate chain, add the `-crl_check_all` parameter.

### 6.14.3 Encrypting a Message

The following command encrypts a plaintext message with the certificate of the recipient:

59 http://www.openssl.org/docs/apps/pkcs7.html

```
openssl smime \
     -encrypt \
     -in plaintext.txt \
     -out smimetext.txt \
     recipient_crt.pem
```

By default, the plaintext is encrypted using the RC2 cipher with a key length of 40 bits. It is recommended to use a strong cipher like AES and a long key, e.g. 256 bits:

```
openssl smime \
     -encryt \
     -aes256 \
     -in plaintext.txt \
     -out smimetext.txt \
     recipient_crt.pem
```

**Note:** In section 6.14.1, the `-certfile` parameter is introduced to include certificate of intermediate Certificate Authorities in the S/MIME message to support the verification process.

### 6.14.4 Decrypting a Message

An encrypted message is easily decrypted if both, the private and the public key, are available. All parameters concerning the cryptographic algorithm are encoded in the ciphertext:

- Decrypt a message by supplying a key pair:

```
openssl smime \
     -decrypt \
     -recip recipient.pem \
     -in smimetext.txt
```

- Decrypt a message by supplying separate public and private keys:

```
openssl smime \
     -decrypt \
     -recip recipient_crt.pem \
     -inkey recipient_key.pem \
     -in smimetext.txt
```

## *6.15 Secure Socket Layer / Transport Layer Security*

Based on X.509 Public Key Infrastructure, two standards provide authentication, data encryption and integrity at the transport layer. The older standard *Secure Socket Layer (SSL)*[60] in its current version 3.0 was developed by Netscape and dates back to 1996. In 1999, it was improved by the Internet Engineering Task Force (IETF)[61] and published as *Transport Layer Security (TLS)*[62]. Although the security is provided at the transport layer

60 http://wp.netscape.com/eng/ssl3/
61 http://www.ietf.org
62 http://www.ietf.org/rfc/rfc2246.txt

and, therefore, is suitable for all communication regardless of the application layer, SSL/TLS is most commonly used to secure HTTP traffic between web servers and clients.

### 6.15.1 Server Socket

The s_server command of the OpenSSL library allows sockets to be created and SSL connections to be initiated upon a client connection[63]:

```
openssl s_server \
     -accept 443 \
     -cert host_cert.pem \
     -key host_key.pem
```

**Note:** If the private key is protected by a passphrase, it can be specified on the command line by the `-pass` parameter (see section 4.5.2).

After a client has successfully connected to the SSL server, the connection can be controlled by the server using the following hotkeys:

| ***Hotkey*** | ***Description*** |
|---|---|
| q | Ends the current connection but accepts a new connection |
| Q | Ends the current connection and exits |
| r | Renegotiate the SSL session |
| R | Renegotiate the SSL session and request a client certificate |
| P | Send random plaintext to the client to cause a protocol exception. It is expected that the client dies. |
| S | Print some information about the status of the session cache |

*Table 4: Hotkeys for openssl s_server*

Alternatively, the server can be asked to offer a HTML page containing status information in reply to any request of a connected client:

```
openssl s_server \
     -accept 443 \
     -cert host_cert.pem \
     -key host_key.pem \
     -www
```

The `s_server` command also provides a very simple web server with SSL support. By specifying the `-WWW` command, pages are resolved to files relatively to the current directory when the command is executed.

**Note:** The `-www` as well as the `-WWW` switches do not offer remote control via the keyboard.

63 http://www.openssl.org/docs/apps/s_server.html

### 6.15.2 Verifying a Certificate

Instead of verifying a certificate against a Certificate Authority, it can be verified by using a web browser:

- Create a SSL/TLS listening socket using the certificate in question:

  ```
  openssl s_server \
       -accept 443 \
       -www \
       -cert host_crt.pem \
       -key host_key.pem
  ```

  You may also need to specify the corresponding private key in order to create the server socket. This can be achieved by entering it interactively or providing the `-pass` parameter.

- Open your favourite web browser and point it to https://localhost:443/.
- Examine the presented certificate.

### 6.15.3 Client Connections

The `s_client` command of the OpenSSL library allows an SSL connection to be established with the specified server and port[64]:

```
openssl s_client \
     -connect localhost:443 \
     -CApath trusts \
     -verify -1
```

The command includes parameters (`-CApath trusts -verify -1`) that force the client to verify the server certificate and terminate the connection if it fails.

**Note:** The `-reconnect` parameter causes `s_client` to reconnect five times using the same session key to test the session caching capabilities of the server.

### 6.15.4 Cipher Lists

The `s_server` as well as the `s_client` command support the `-cipher` parameter which allows a list of supported or preferred cipher suites (the *cipher list*) to be specified, respectively.

A cipher list can either be build manually be building a colon-separated list of cipher suites from the output of the `openssl ciphers -v` command or by specifying a colon-separated list of *cipher strings* which define matching parameters for cipher suites.

The following table contains valid and implemented cipher strings:

64 http://www.openssl.org/docs/apps/s_client.html

| ***Cipher String*** | ***Description*** |
|---|---|
| DEFAULT | ALL:!ADH:RC4+RSA:+SSLv2:@STRENGTH<br>(also: COMPLEMENTOFDEFAULT) |
| ALL | All cipher suites but eNULL<br>(also: COMPLEMENTOFALL) |
| HIGH | Key length > 128 bits |
| MEDIUM | Key length = 128 bits |
| LOW | Key length < 128 bits |
| eNULL, NULL | Cipher suites without any encryption |
| AES<br>DES<br>3DES<br>RC2<br>RC4<br>IDEA | Cipher suites using the specified encryption algorithm |
| SHA<br>SHA1<br>MD5 | Cipher suites using the specified message digest algorithm |
| TLSv1<br>SSLv2<br>SSLv3 | Cipher suites specified by the standards |
| aDH | Anonymous DH |
| DH | DH authentication including aDH |
| aRSA<br>aDSS, DSS | Authentication using the specified algorithm |
| kRSA, RSA | RSA key exchange |
| kEDH | Ephemeral DH key agreement |
| EXP, EXPORT<br>EXPORT40<br>EXPORT56 | Export encryption algorithms (all or with a specific key length) |

*Table 5: Cipher Strings for openssl s_server and s_client*

Individual cipher strings in a cipher list, can be prepended by one of three characters to modify their meaning:

- By prepending an exclamation mark ("!"), the matching cipher suites are not used and cannot be added by later cipher strings.

- Matching cipher suites can be assigned a lower priority by prepending a plus ("+") and, thereby, moving them to the end of the cipher list.
- Similar to the exclamation mark, a prepended minus ("-") removed the matching cipher suites from the list but allows them to be added again.

By adding the `@STRENGTH` command to the list, the cipher suites that have been matched so far are sorted by their key length.

**Note:** To establish unencrypted SSL connections with `s_server` and `s_client`, use the following parameters on both ends: `-cipher NULL-SHA`. The `NULL` cipher is not included in the default cipher list so that an unencrypted channel cannot be opened by using the parameter on one endpoint only.

### 6.15.5 Client Certificates

Although the client is assured of the server's authenticity by the presented certificate, the server usually does not know whether the connecting client or user is authorized to access the served content. Therefore, the server can require the client to present a certificate that is verified against a collection of trusted certificates:

- Create the server socket:

```
openssl s_server \
     -accept 443 \
     -cert Bob\bob.pem \
     -www \
     -Verify 0 \
     -Capath Bob\trusts
```

- Connect the client:

```
openssl s_client \
     -connect localhost:443 \
     -cert Alice\alice.pem \
     -verify -1 \
     -Capath Alice\trusts
```

Using the presented command, both communication endpoints are required to successfully authenticate before data can be exchanged.

### 6.15.6 Handshake

After the client has connected to a SSL/TLS server socket, it sends the clientHello message and expects to receive the serverHello message followed by the server certificate (optional), the server's shared secret for bulk encryption (optional) and the request for a client certificate (optional). The client answers it by verifying the server's certificate (if applicable), sending its certificate (optional) and sending its shared secret for bulk encryption.

**Further Study:** The internals of the SSL/TLS handshake can be analyzed by the `-state` parameter causing the `s_server` and `s_client` command to output helpful information. The `-msg` parameter shows the content of the messages transmitted by the protocol. If the detailed behaviour of the protocol needs to be analyzed, the `-debug` parameter displays hexadecimal dumps of the individual read and write operations. The opposite of the mentioned parameters is achieved by the `-quiet` parameter.

**Further Study:** When the handshake has been performed, client and server have agreed upon a set of parameters to be used in the session. This information is printed by the `s_server` command in a PEM-encoded structure enclosed by `BEGIN/END SSL SESSION PARAMETERS`. It is decoded by the following command: `openssl sess_id -text -noout -in sess_id.pem`. This human-readable output is also printed by the `s_client` command.

**Further Study:** By default, the connection and read timeouts are disabled by the `s_server` command. To enable them, add the `-timeout` parameter.

**Further Study:** The SSL/TLS standard only allows TCP sockets and connections to be created. By adding the `-dtls1` parameter, the Datagram Transport Layer Security standard is used creating a UDP socket with the `s_server` command and connecting to it with the `s_client` command.

### 6.15.7 Diffie-Hellman-Merkle Parameters

The `s_server` and `s_client` commands of the OpenSSL library make use of default parameters for the Diffie-Hellman-Merkle key exchange algorithm which are hard coded into the OpenSSL library. Obviously, this may pose a security threat because some of the initial values are publicly known. In the following commands, a file with pre-created parameters for the Diffie-Hellman-Merkle key exchange algorithm is used (refer to section 4.2 for instructions how to create a DH parameter file):

- Use custom DH parameters on the server side:

```
openssl s_server \
     -dhparam host_dh.pem
```

- Use custom DH parameters on the client side:

```
openssl s_server \
     -accept 443 \
     -cipher kEDH \
     -dhparam Alice\alice_dh.pem \
     -nocert \
     -cert Alice\alice.pem
```

**Note:** The `-cipher` parameter accepts a cipher list which was introduced in section 6.15.4.

**Note:** To ensure that the hard coded DH parameters are not used, add `-no_dhe -no_ecdhe` to your command line.

### 6.15.8 Certificates for Multiple Virtual Hosts

When certificates are implemented to provide a secure service, it is often necessary to secure several services using a single certificate. This effectively reduces the management effort and costs for certificate enrolment and maintenance. Due to the fact that individual services are often accessed via separate host names (virtual hosts on a web server), it is possible to embed several names in certificates. For the sake of simplicity, this section assumes that a server certificate for the DNS names is used.

Apart from the Common Name, a X.509 extension can be added to a certificate which specifies an alternative name for the subject of a certificate. The following command demonstrates the inclusion of more than one Common Name in a single Certificate Signing Request:

```
openssl req \
     -config openssl.config -extensions extensions_root_CA \
     -x509 \
     -new \
     -subj "/CN=hostname1/CN=hostname2"
     -key test_key.pem \
     -out test_crt.pem
```

**Warning:** Unfortunately, the Mozilla Firefox[65] web browser does not correctly process a certificate containing multiple Common Names; only the last Common Name is recognized. Microsoft Internet Explorer correctly recognized and processes multiple Common Names.

Unfortunately, embedding one or more alternative names for the subject cannot be achieved on the command line but requires the configuration file to be modified to include the `subjectAltName` parameter. This change is introduced either by adding a new section or by modifying an existing section containing the extensions to be added to a certificate. The following syntax is supported by the OpenSSL library:

- Move the email address from the Common Name to the alternative name:

  ```
  subjectAltName=email:move
  ```

- Copy the Common Name to the alternative name:

  ```
  subjectAltName=DNS:copy.commonName
  ```

- Add a custom email address:

  ```
  subjectAltName=email:user@domain.tld
  ```

- Add a custom DNS hostname:

  ```
  subjectAltName=DNS:hostname
  ```

- Add an IPv4 or IPv6 address:

  ```
  subjectAltName=IP:1.2.3.4
  ```

65 http://www.mozilla.com/firefox/

```
subjectAltName=IP:1::2
```

- Add an Uniform Resource Indicator (URI):

```
subjectAltName=URI:http://my.url.here/
```

The alternative names can also be a comma-separated list of these items.

**Note:** The Mozilla Firefox web browser as well as Microsoft Internet Explorer correctly recognize and process the subjectAltName extension in certificates.

**Note:** It is recommended to repeat a certificate's Common Name as the alternative subject name.

**Note:** When creating a Certificate Signing Request with X.509 extensions, the Certificate Authority needs to copy these extensions from the CSR into the certificate during enrolment. Please refer to section 6.6.6 for details.

## *6.16 Risks and Issues*

The attentive reader will have noticed that several issues are present in the design of Public Key Infrastructures. You need to be aware of these issues to decide whether this type of PKI is an option for you. Each of the following sections focuses on one risk or issue that was identified and provide additional insight.

In comparison to specialized devices, personal computers are inherently unsuited for any kind of private communication. The hardware design allows for several types of eavesdropping and the software is not designed to secure data but to enable users to perform their work. Cryptography strives to make up for these shortcomings.

### 6.16.1 Practices of the Certificate Authority

A Certificate Authority defines its practices in the Certificate Practices Statement, e.g. how the identity information of an entity is verified before a certificate is enrolled. Most customers choose a public Certificate Authority based on the pricing model. Although money is always short, you should be worried about the security of the Public Key Infrastructure which you are about to implement or participate in. You should worry about the contents of the CPS and ask yourself the following questions:

1) Does the Certificate Practices Statement meet your needs?
    a) Is the identity information of the applicant supposed to be verified thoroughly?
    b) Do the offered key lengths and the supported certificate lifetime meet your security and performance requirements?
    c) Does the CA define a feasible certificate renewal process?
    d) Is a Certificate Revocation List supposed to be published in a timely fashion? Does the CA offer the Online Certificate Status Protocol?

2) Does the Certificate Authority follow its Certificate Practices Statement or, rather, can it be trusted to follow its CPS?
3) Are you supported or even encouraged when implementing your own Subordinate Certificate Authority? What obstacles need to be overcome to qualify for this role?

A Public Key Infrastructure is based on trust and, therefore, your first step should be to ensure that the issuer of your certificate or certificates can be trusted. Otherwise, you will be introducing the first security breach.

### 6.16.2 Trust Issues

A Public Key Infrastructure is based on cryptography, symmetric as well as asymmetric, and it is held together by the trust that is placed in a Certificate Authority to follow its Certificate Practices Statement.

But a Public Key Infrastructure implies that a user can only partially trust a Subordinate Certificate Authority because it cannot operate stand-alone and needs a superior Certificate Authority. In the end, only a Root CA is able certify that its Subordinate Certificate Authorities are allowed to issue and manage certificates. An entity is required to place total trust in the Root CA that certified the corresponding Subordinate Certificate Authorities.

Remember that a Root CA is based on a self-signed certificate which has inherent security issues because an entity either needs to rely on its vendor to supply an authentic root certificate or needs to verify the authenticity on its own. The latter is not a trivial process and prone to errors and attacks. Therefore, it is important to be aware of that the integrity and the authenticity of a collection of root certificates are of utmost importance in a Public Key Infrastructure.

### 6.16.3 Security of the Private Key

When implementing public key cryptography, as in Public Key Infrastructures, the private key needs to be kept safe from unauthorized access and usage. This requirement raises three important questions:

4) Will you be able to properly secure your own private keys?
5) Will you be able to trust authorized entities to keep their private keys secure?
6) Will you be able to trust other entities to handle the private key securely, like administrators or backup operators?

Only in a tightly controlled environment you can be at least moderately sure that the above questions can be answered positively. Consequently, you can hardly be sure that the private keys of all participating entities remain uncompromised.

During certificate enrolment, this fact causes severe problems because the Certificate Authority cannot be sure that the applicant is the holder of the corresponding private key. Although a CA can provide means to generate a private key during enrolment, as soon as the private key leaves the Certificate Authority's influence its security is unknown. In this scenario, the applicant also needs to be aware of the additional insecurity of allowing the private key to be generated by the Certificate Authority.

### 6.16.4 Certificate

It is often assumed that a certificate provides entity authentication and ensures that the requested remote entity was reached. Instead, a certificate only binds the identity of an entity to a specific public key and, therefore, only associates an entity with its public key. The connection from a public key to an entity is much looser than to its identity information.

Consequently, a certificate only ensures that a specific identity was used but does not allow an assumption about the security of the corresponding private key. It does not provide any assurance whether …

- … the private key was available to other entities than the owner at the time of enrolment.
- … the private key is available to other entities than the owner at the time of invocation.
- … a message was signed or encrypted by the owner of the private key.
- … a message was signed or encrypted with the implicit or explicit consent of the owner of the private key.

Such a connection between an entity and a private key is probably impossible because modern computers cannot provide an absolute level of security which ensures that a certain piece of information is safe from usage by an unauthorized entity.

### 6.16.5 Revocation

In theory, a certificate needs to be revoked as soon as it looses its trustworthiness when the private key is lost or compromised. From that moment on, the certificate may not be used for public key cryptography. In real life, this process depends on many components to operate correctly:

- Has the owner detected that the private key has been compromised or lost?
- If the private key is compromised, how much time passes until this is detected?
- Has the owner acted swiftly and revoked it after the private key has been compromised?
- Does the Certificate Authority generate a new Certificate Revocation List after every revocation or only on a regular schedule?
- Do involved entities check the Certificate Revocation List distribution points of certificates or use the Online Certificate Status Protocol before accepting and using a certificate?

**Warning:** The consequences of a compromised certificate are severe if the owner is a Certificate Authority. Since the old private key cannot be used, the corresponding certificates need to be revoked as well as all certificates that have been issued.

### 6.16.6 Non-Repudiation

Non-repudiation is commonly attributed to Public Key Infrastructures because it is assumed that an entity's private key remains uncompromised. Unfortunately, the security of the private key is not absolute and, therefore, susceptible to a range of incidents some of which due to attacks and others due to improper use.

In a Public Key Infrastructure, entities assume that the verification of a certificate against a chain of trusted certificates is equivalent to the identification of the entity invoking the private key. But like a handwritten signature, its digital counterpart only provides a strong indication which entity is invoking the private key. Therefore, both are prone to the same kinds of attacks:

- The handwritten signature was forged. The digital signature was created by invoking a stolen private key.
- The signature is not forged but the owner has participated in its generation unknowingly.
- The signature is not forged but the owner was forced to participate in its generation.

### 6.16.7 Roles

The identity information of a certificate is based on X.509 directories which define a tree-like structure of the indexed entries. The position of an entry is described by the Distinguished Name containing the Common Name which identifies the entity in the represented geographical or organizational structure.

The central concept of identifying an entity by its Common Name also introduces a major disadvantage because, in real life, a single entity can represent several roles each of which may need to be expressed by a separate key pair. Due to the fact that this requirement is not inherently supported by X.509 directories, roles need to be encoded in the Common Name causing the Distinguished Name to be abused to support roles.

### 6.16.8 Ignorance of Users

Certificates are widely used to secure SSL connections by providing authentication and integrity to one, some or all endpoints depending on who offered a certificate. When considering web shops, the end user is confronted with a system that he does not understand but is required to trust its security because, when purchasing an item, he most likely reveals private data like credit card information to the server-side application.

The following questions have been collected to demonstrate the lack of knowledge of an end user and the possible effects. This list focuses on server certificates.

- Is the user able to determine whether he is accessing a site over an authenticated and encrypted connection utilizing certificates?
- Does the user have a basic understanding what a certificate is and does?
- Does the user recognize the importance of a certificate warning?

  When the user is presented with such a warning, it usually contains three bullets which can turn to red, meaning that the authenticity, confidentiality or integrity of the certificate may have been compromised.

  - If a certificate cannot be verified, a warning is issued. This is caused by one or more certificates in the chain that were required in order to verify the signature of the certificate in question. This hints toward a certificate which was presented to deceive the user into revealing critical information.
  - If the Common Name of the certificate does not match the server name, a warning is issued. The user might have been redirected to a different server than he intended to reach.
  - If the presented certificate has outlived it validity, a warning is issued. Obviously, the owner has failed to renew the certificate. Consequently, the Certificate Authority was not able to ensure that the certificate still binds the public key to the included identity information.

Due to this ignorance, the user is exposed to Man-in-the-Middle (MitM) attacks.

### 6.16.9 Limitation to Transport Layer

A Public Key Infrastructure allows for authentication and integrity of the exchanged data and provides some degree of protection against attacks. For the end user, the security of the disclosed information inside an unknown system is critical.

It is often criticized that SSL/TLS, a network protocol built on X.509 PKI, does not extend to the application level. This is a rather limited view because PKI can as well be used to certify the integrity of an application. But this requires new authorities to be implemented which define standards for the security of personal information, verify that the standards are honoured and issue certificates which attest to this state. Of course, this introduces new issues that need to be addressed but it is unfair to accuse a communication protocol for the transport layer to not extend to higher layers.

Ultimately, the user needs to trust the remote site to handle the personal information in a secure manner.

## *6.17 Summary*

Are there unresolved issues in X.509 Public Key Infrastructures? Yes.

Do system architects need to be aware of these issues? Yes.

Is there an alternative to X.509 Public Key Infrastructures? Possibly yes.

Do we have to stop using X.509 Public Key Infrastructures? No.

# 7 OpenPGP Public Key Infrastructures

Contrary to the design of an OpenSSL based Public Key Infrastructure, *GNU Privacy Guard (GnuPG)*[66] implements distributed trust relationships called a *web of trust*. Using this trust model does not require special design components because every participant represents an authority and manages a collection of trust relationships. Due to the fact that a web of trust consists of equivalent entities, certificates may contain several signatures to allow for extensive cross-signing. In addition, GnuPG enables entities to take on several roles. Due to these features, GnuPG offers extensive facilities for the management of keys and trusts.

GnuPG is a fully OpenPGP[67] compliant implementation which is not based on the parented IDEA algorithm like the original *Pretty Good Privacy (PGP)*[68] implementation.

## 7.1 Special Features

Apart from typical features of Public Key Infrastructures like confidentiality, authenticity and integrity, GnuPG offers a number of special features which are not found in X.509 Public Key Infrastructures:

**User Identities.** A single entity is often required to take on several *roles* and perform key operations which need to be isolated by separate key pairs. GnuPG supports entities to manage and use more than one key pair and also allows several *user identities (UID)* to be added to a single key pair (see Illustration 17).

**Key Rings.** Entities can organize and manage collections of keys called *key rings*. There are two types of key rings to separate *secret keys* and *public keys*.

**Multiple Signatures.** A single public key can be signed by more than one private key (see Illustration 17) because, in a web of trust, every entity represents an authority and may attest to the validity of a key.

**Trusts Levels.** Each public key in a key ring needs to be assigned a *trust level* to define the extent to which the owner is trusted. Based on transitive relationships, newly imported keys can be automatically assigned a trust level. Trust levels also define which entities are considered trusted authorities and, therefore, enable successful authentication.

**Sub keys.** A single secret key consists of a *master key* and, optionally, a list of *sub keys* (see Illustration 17) each of which can be used for either encryption or signing. This allows for the separation of tasks, does not expose the master key and increases its lifetime.

**Compression.** Messages may be compressed using different algorithms after encryption or attaching a signature.

66 http://www.gnupg.org
67 http://www.ietf.org/rfc/rfc2440.txt
68 http://www.pgp.com, http://www.pgpi.org

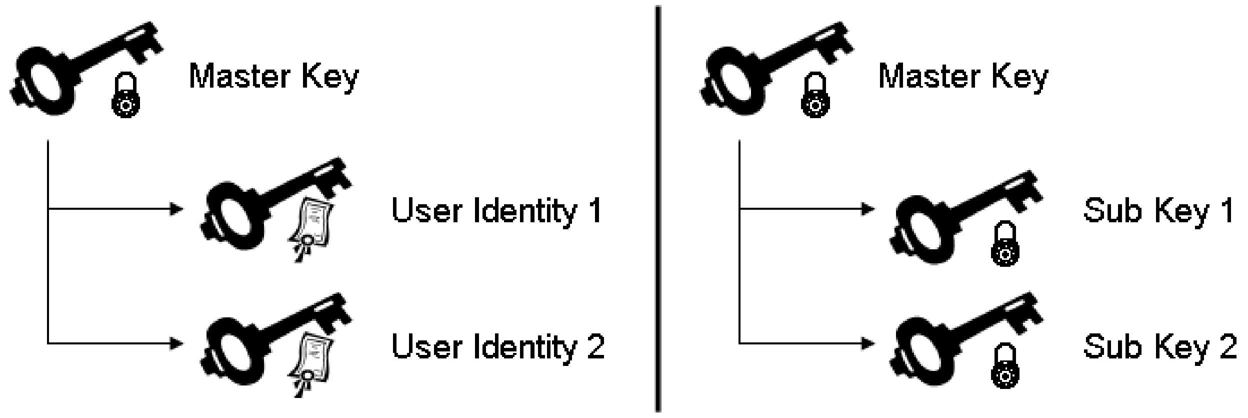

*Illustration 17: OpenPGP Features*

## 7.2 Configuration and Invocation

To evaluate GnuPG in an isolated environment, all operations are restricted to special key rings in the current directory:

```
gpg \
     --homedir .
```

This special parameter also forces GnuPG to read its configuration file `gpg.conf` from the specified directory and, effectively, allows for the evaluation of new parameters and features. The following example shows a typical configuration file:

```
no-greeting
no-secmem-warning
armor
no-emit-version

s2k-cipher-algo AES256
s2k-digest-algo SHA1

cipher-algo AES256
digest-algo SHA1
compress-algo bzip2
cert-digest-algo SHA1

default-sig-expire 0
no-ask-sig-expire
default-cert-expire 5y
no-ask-cert-expire

auto-check-trustdb

# list options
```

```
list-options show-keyserver-urls show-uid-validity show-unusable-uids
     show-unusable-subkeys show-sig-expire
# verify options
verify-options show-keyserver-urls show-uid-validity show-unusable-uids
```

The armor parameter forces output to be ASCII encoded while the `no-emit-version` parameter suppresses the inclusion of the version of the GnuPG command line tool. A user defined comment is added to the output by the `--comment` parameter.

## 7.3 Basic Key Management

GnuPG does not utilize Distinguished Names to build a hierarchy of keys because the tree-like structure of X.500 directories is not compatible with a web of trust due to the need for cross-signing between entities. In addition, an entity's identity information is an inherent component of a secret key specified by a name, an email address and an optional comment.

At the time of this writing, GnuPG does not support a fully unattended invocation, so that the generation and revocation of key pairs needs to be performed interactively:

- Generate a new key pair: You are required to specify a key type, length, lifetime and identity information.

```
gpg \
     --homedir . \
     --gen-key
```

- Revoke a secret key identified by `NAME`: You are required to specify a reason for the revocation.

```
gpg \
     --homedir . \
     --gen-revoke NAME
```

Public and secret keys are stored and managed in individual files called key rings so that the management needs to be performed separately:

- List private keys:

```
gpg \
     --homedir . \
     --list-secret-keys
```

- List public keys:

```
gpg \
     --homedir . \
     --list-public-keys
```

- Delete a secret key identified by `NAME`:

```
gpg \
     --homedir . \
     --delete-secret-keys NAME
```

- Delete a public key identified by `NAME`:

```
gpg \
     --homedir . \
     --delete-key NAME
```

Similar to the OpenSSL based certificate, GnuPG defines a fingerprint which allows a key specified by `NAME` to be identified unambiguously[69]:

```
gpg \
     --homedir . \
     --fingerprint NAME
```

## *7.4 Backup*

Backing up a collection of public and secret keys is achieved easily because GnuPG operates on a well-defined set of files. They can be moved, copied and restored to implement an individual backup scheme.

Although the aforementioned files represent the public as well as the secret key ring, a simple file based backup does not protect against corruption.

### 7.4.1 Owner Trust

The following commands demonstrate the backup and restore process of the data contained in the `trustdb.gpg` file:

- Backup owner trust to `ownertrust.txt`:

```
gpg \
     --homedir . \
     --export-ownertrust \
     >ownertrust.txt
```

- Restore owner trust from `ownertrust.txt`:

```
gpg \
     --homedir . \
     --export-ownertrust ownertrust.txt
```

Again, this file can be included in a file based backup of the GnuPG installation.

### 7.4.2 Keys

Both public and secret keys are stored in binary key rings but the individual keys can be extracted and saved in an *ASCII armor*, a Base64 encoding. The representation is similar

69 Of course, this only means that the specified key pair can be identified unambiguously with a high probability but not with absolute certainty.

to the PEM format of the OpenSSL library. Due to the public and secret key rings, separate commands are required to extract their contents:

- Export a public key: If the `NAMES` parameter is omitted, all keys in the ring are exported.

```
gpg \
     --homedir . \
     --export --armor \
     --output pub.gpg --yes \
     [NAMES]
```

- Export a secret key: If the `NAMES` parameter is omitted, all keys in the ring are exported.

```
gpg \
     --homedir . \
     --export-secret-keys --armor \
     --output sec.gpg --yes \
     [NAMES]
```

GnuPG determines the appropriate key store without the need to implement separate import commands for public and secret keys:

```
gpg \
     --homedir . \
     --import \
     key.gpg
```

## 7.5 Advanced Key Management

In addition to the basic tasks for key management which were presented in section 7.3, GnuPG supports an interactive mode to modify public and secret keys:

```
gpg \
     --homedir . \
     --edit-key UID
```

In this mode, a series of commands is provided to perform the required tasks on the key. A full list can be retrieved by the command help.

- List the details of the key: `list`
- Display the fingerprint of the key: `fpr`
- Check the signatures of the key: `check`
- Change the passphrase of the key: `passwd`
- Assign a trust level to the key: `trust`

Trust levels allow new keys to be trusted automatically if it is signed by a fully trusted key or by three marginally trusted keys. The length of the path from your own to the new key is also taken into account reducing the trust level of a new key.

There are several commands which operate on the master key, one of its sub keys or one of its user identities. After entering the edit mode, no sub key and no user identity is selected and the following operations are performed on the master key:

- Revoke the master key: `revkey`
- Change the expiration date of the master key: `expire`

### 7.5.1 User Identities

The support of more than one user identity allows a single key (the master key and its sub keys) to be used with several identities (see Illustration 17). A newly generated secret key contains a single user identity; more can be added by the adduid command.

All other operations need to be performed on a selected user identity. A UID is selected by the uid N command in which N is the number of the UID in question. This number can be retrieved from the output of the list command. By executing uid 0 (zero), all user identities are deselected.

- Define the primary user identity: `primary`
- Revoke the selected user identity: `revuid`

A by-product of revoking a user identity is the loss of all signatures to the concerned UID.

### 7.5.2 Sub Keys

Similar to the management of user identities, several commands are provided to manage sub keys of a master key (see Illustration 17). These sub keys can be used to perform separate tasks or support multiple roles by single entity. A new sub key is added by the addkey command.

All other operations need to be performed on a selected sub key by the key N command in which N is the number of sub key to be selected. This number needs to be derived from the output of the list command. By executing key 0 (zero), all keys are deselected.

- Delete the selected sub key: `delkey`
- A key can be revoked by the `revkey` command. If no sub key is selected, this command revokes the master key and, therefore, also revokes its sub keys.

### 7.5.3 Signatures

To build a web of trust, user identities need to be signed. Please refer to section 7.5.1 on how to select user identities.

A user identity can be signed locally to express trust which is not to be shared publicly and traditionally which is shared by default when sending the key to a key server:

- Sign the selected user identity locally: `lsign`
- Sign the selected user identity: `sign`

- Sign the selected user identity irrevocably: `nrsign`
- Sign the selected user identity and assign marginal or full trust: `tsign`
- Revoke a signature of the selected user identity: `revsig`

## 7.6 Message Processing

Although some invocations of GnuPG cannot be called in an unattended fashion, the passphrase can be read from Standard Input. Unfortunately, all other input needs to be provided via files[70]:

```
Type passphrase.txt \
| gpg \
     --passphrase-fd 0 \
     ...
```

When operations on messages are to be performed, a key must be selected. This can be achieved by specifying a command line parameter for every invocation of GnuPG. Due to the fact that users usually possess a primary key which is used for most operations, the GnuPG configuration file can be extended to include a parameter specifying the default key. In addition, it is a reasonable decision to force all content to be encrypted with the sender's key as well:

```
default-key NAME
default-recipient-self
```

The output of GnuPG message operations like signing and encrypting is in the OpenPGP format which is similar but incompatible to S/MIME. In the following examples, the ASCII armoured output is chosen.

### 7.6.1 Signatures

GnuPG supports three methods to create a signature. The following commands demonstrate these different flavours by utilizing the default digest algorithm defined by digest-algo in the configuration file.

- By specifying the `--sign` parameter, the message and the attached signature are compressed utilizing the default compression algorithm defined by `compress-algo` in the configuration file:

```
gpg \
     --homedir . \
     --sign \
     --armor \
     --output FILE.asc \
     FILE
```

- Attach a signature to a non-binary message without compression:

70 You do not need to transfer your key ring to the host on which the files are residing. Instead process the files locally via an encrypted tunnel:
`ssh USER@HOST 'cat FILE1' | gpg ... | ssh USER@HOST 'cat - > FILE2'`

```
gpg \
     --homedir . \
     --clearsign \
     --armor \
     --output FILE.asc \
     FILE
```

- The following command creates a detached signature which remains separate from the signed message. This is especially useful to sign binary data, e.g. compressed source code, and distribute the signature in a separate file:

```
gpg \
     --homedir . \
     --detach-sign \
     --armor \
     --output FILE.asc \
     FILE
```

For the verification of a signature, GnuPG needs the message and the signature:

- In the case of an attached signature, only a single file is needed to verify the signature:

```
gpg \
     --homedir . \
     --verify \
     FILE.asc
```

- If a detached signature is to be verified, both the message (`FILE`) and the signature (`FILE.asc`) are to be supplied:

```
gpg \
     --homedir . \
     --verify \
     FILE.asc FILE
```

### 7.6.2 Encryption

GnuPG allows messages to be encrypted symmetrically as well as asymmetrically. The type of encryption and its parameters like the utilized algorithm are encoded in the resulting OpenPGP message to allow for easy decryption:

- Encrypt a message contained in `FILE` using asymmetrical cryptography:

```
gpg \
     --homedir . \
     --encrypt \
     --recipient NAME \
     --yes \
     FILE
```

- Encrypt a message contained in `FILE` using the default cipher specified by `cipher-algo` in the configuration file:

```
gpg \
     --homedir . \
     --symmetric \
     --yes \
     FILE
```

- Decrypt a message in `FILE`:

```
gpg \
     --homedir . \
     --decrypt FILE
```

## 7.7 Key Servers

Due to the fact that every participant of an OpenPGP PKI represents an authority and is enabled and encouraged to sign keys, the publication of a signed public key is very important. To resolve issues with public key distribution, key servers accept, store and serve public keys. A collection of favourite key servers can be defined in the configuration file. In addition, a list of options can be defined for search, import and export operations:

```
# key server url
keyserver hkp://subkeys.pgp.net
# search options
keyserver-options include-revoked include-disabled honour-keyserver-url
     include-subkeys auto-key-retrieve
# import options
import-options no-import-local-sigs no-merge-only import-clean-sigs
     import-clean-uids
# export options
export-options no-export-local-sigs export-attributes no-export-minimal
     export-clean-sigs export-clean-uids
```

After generating and customizing a secret key, its public component should be sent to a publicly available key server to enable other entities to verify signatures and send encrypted messages. The following command demonstrates sending the public key component of the secret key denoted by `NAME` to the favourite key server:

```
gpg \
     --homedir . \
     --send-keys NAME
```

Before an operation with another entity's public key can be performed, it needs to be retrieved from a key server. This is achieved either automatically by specifying the `keyserver-option auto-key-retrieve` in the configuration file or manually executing the following command for the public key denoted by a list of key ids:

```
gpg \
     --homedir . \
     --recv-keys KEYIDS
```

Keys should be refreshed regularly after having been retrieved to the local device because they may have been updated:

```
gpg \
     --homedir . \
     --refresh-keys [KEYIDS]
```

## 7.8 Managing Trust

Apart from the interactive editing mode presented in section 7.5, trust can also be managed from the command line:

- List all public keys with signatures:

  ```
  gpg \
       --homedir . \
       --list-sigs
  ```

- Sign the key denoted by `NAME` using the default key:

  ```
  gpg \
       --homedir . \
       --sign-key NAME
  ```

- Sign the key denoted by `NAME` using the default key locally:

  ```
  gpg \
       --homedir . \
       --lsign-key NAME
  ```

Due to expiration dates set on keys and signatures, the trust database needs to be updated regularly. This can either be enforced by the `auto-check-trustdb` command in the configuration file or by explicitly on the command line:

- The following command checks the trust database for expired keys and signatures and updates the owner trust definitions accordingly:

  ```
  gpg \
       --homedir . \
       --check-trustdb
  ```

- The following command performs the same operations on the trust database as the previous command but, in addition, forces the user to fill empty owner trust definitions:

  ```
  gpg \
       --homedir . \
       --update-trustdb
  ```

## 7.9 Summary

This appendix about GNU Privacy Guard demonstrates the significant differences between hierarchical and distributed Public Key Infrastructures. In a web of trust, which is implemented by GnuPG, every entity represents an authority, trusts selected entities and consider some of them to be authorities as well.

The distributed trust model causes a much larger number of trust relationships to be established because entities are able to participate more actively and shape the web of trust. This introduces a major threat to an OpenPGP PKI because it is harder to control.

# 8 Comprehensive Examples

The previous chapters covered the theoretical part of Public Key Infrastructures with lots of examples showing how specific tasks are performed, this chapter provides the glue to build a comprehensive example that demonstrates most of the covered tasks and commands.

It is assumed that Alice and Bob are required to exchange one or more messages that need to be secured. In addition, a Certificate Authority enables them to easily create trust relationships.

## *8.1 Common Steps*

This section describes common steps that are needed by both examples in the following sections. Choose or create an empty directory, the base directory, in which one or both examples will be performed.

### 8.1.1 Preparations

The base directory for the examples represents the public channel which is utilized for communication. Each participating party, the Certificate Authority, Alice and Bob, with their respective private data is represented by a subdirectory of the base directory:

```
md \
     CA Alice Bob
```

Now, save the following data to the file `CA\openssl.config` in the base directory:

```
[ca]
default_ca      = certificate_authority

[certificate_authority]
dir             = .
certs           = $dir/crt
new_certs_dir   = $dir/crt
crl_dir         = $dir/crl
database        = $dir/index
certificate     = $dir/ca_crt.pem
serial          = $dir/serial
crl             = $dir/ca_crl.pem
private_key     = $dir/key/ca_key.pem
RANDFILE        = $dir/key/.rand
default_crl_days= 30
default_days    = 365
default_md      = sha1
x509_extensions = extensions_user
preserve        = no
```

```
policy          = policy_match

[policy_match]
commonName               = supplied
countryName              = optional
stateOrProvinceName      = optional
localityName             = optional
organizationName         = optional
organizationalUnitName   = optional
emailAddress             = optional

[policy_anything]
commonName               = supplied
countryName              = optional
stateOrProvinceName      = optional
localityName             = optional
organizationName         = optional
organizationalUnitName   = optional
emailAddress             = optional

[req]
default_bits             = 2048
default_keyfile          = ./key/ca_key.pem
default_md               = sha1
distinguished_name       = req_distinguished_name

[req_distinguished_name]
commonName                      = Common Name
commonName_default              = example CA
countryName                     = Country
countryName_default             = DE
#countryName_min                 = 2
#countryName_max                 = 2
stateOrProvinceName             = State or province
stateOrProvinceName_default     = NRW
localityName                    = Locality
localityName_default            = Cologne
organizationName                = Organization
organizationName_default        = example GmbH
organizationalUnitName          = Organizational Unit
organizationalUnitName_default = IT
emailAddress                    = Email address
emailAddress_default            = certmaster@example.com
```

```
[extensions_root_CA]
#nsCaRevocationUrl     = http://www.example.com/root_ca/ca_crl.pem
basicConstraints      = critical, CA:true
nsCertType            = objCA,emailCA,sslCA
#crlDistributionPoints = URI:http://www.example.com/root_ca/ca_crl.pem
#subjectAltName        = email:certmaster@example.com
subjectKeyIdentifier  = hash
#nsCaPolicyUrl         = "http://www.example.com/root_ca/policy/"

[extensions_subordinate_CA]
#nsCaRevocationUrl     = http://www.example.com/sub_ca/ca_crl.pem
basicConstraints      = critical, CA:true
nsCertType            = objCA,emailCA,sslCA
#crlDistributionPoints = URI:http://www.example.com/sub_ca/ca_crl.pem
#subjectAltName        = email:certmaster@example.com
subjectKeyIdentifier  = hash
#nsCaPolicyUrl         = "http://www.example.com/sub_ca/policy/"

[extensions_user]
nsCertType             = server,client,email
#crlDistributionPoints  = URI:http://www.example.com/crl.shtml
#nsCaPolicyUrl          = http://www.example.com/policy.html
#subjectAltName         = email:certmaster@example.com
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid,issuer:always
keyUsage               = digitalSignature, nonRepudiation,
     keyEncipherment, dataEncipherment
basicConstraints       = CA:false
```

### 8.1.2 Set up the CA

In the `CA` subdirectory, set up a Root Certificate Authority by creating the necessary directories, initializing the database and generating a private key and with a self-signed certificate:

```
md key crt crl
type Nul >index
echo 01>serial
openssl req \
     -config openssl.config -extensions extensions_root_CA \
     -x509 \
     -newkey rsa:1024 \
     -nodes \
     -subj "/CN=Test CA/C=DE/ST=NRW/L=Cologne/O=Test GmbH/OU=IT" \
     -keyout key\ca_key.pem \
     -out ca_crt.pem
```

**Further Study:** View the self-signed certificate (see section 6.6.5) and note the Distinguished Names of subject and issuer.

### 8.1.3 Generating Private Keys

Before certificates for the participating parties can be created, private keys for Alice and Bob need to be generated:

- Generate a RSA private key in Alice's subdirectory:

```
openssl genrsa \
     -out alice_key.pem 1024
```

- Generate a RSA private key in Bob's subdirectory:

```
openssl genrsa \
     -out bob_key.pem 1024
```

**Further Study:** View the private keys in human-readable form (see section 4.5.5).

### 8.1.4 Obtain a Certificate

In this section, we will generate Certificate Signing Requests for Alice as well as Bob, transfer them to the Certificate Authority, create the certificates and retrieve the certificates from the CA. In addition, we will create a collection of trusted certificates for both parties.

- Execute the following commands from Alice's directory:

```
openssl req \
     -new \
     -key alice_key.pem \
     -subj "/CN=Alice/C=DE/ST=NRW/L=Cologne/O=Test GmbH/OU=IT" \
     -out alice_csr.pem
copy alice_csr.pem ..\CA\
cd ..\CA
openssl ca \
     -config openssl.config \
     -batch \
     -in alice_csr.pem -out alice_crt.pem
del alice_csr.pem
move alice_crt.pem ..\Alice\
cd ..\Alice
type alice_key.pem alice_crt.pem >alice.pem
md trusts
```

```
copy ..\CA\ca_crt.pem trusts\
cd trusts
openssl x509 \
     -hash -noout \
     -in ca_crt.pem
```

The output of the last command is represented by `HASH`.

```
move \
     ca_crt.pem HASH.0
```

- Execute the following commands from Bob's directory:

```
openssl req \
     -new \
     -key bob_key.pem \
     -subj "/CN=Bob/C=DE/ST=NRW/L=Cologne/O=Test GmbH/OU=IT" \
     -out bob_csr.pem
copy bob_csr.pem ..\CA\
cd ..\CA
openssl ca \
     -config openssl.config \
     -batch \
     -in bob_csr.pem -out bob_crt.pem
del bob_csr.pem
move bob_crt.pem ..\Bob\
cd ..\Bob
type bob_key.pem bob_crt.pem >bob.pem
md trusts
copy ..\CA\ca_crt.pem trusts\
cd trusts
openssl x509 \
     -hash -noout \
     -in ca_crt.pem
```

The output of the last command is represented by `HASH`.

```
move \
     ca_crt.pem HASH.0
```

**Further Study:** View the Certificate Signing Request in human-readable form (see section 6.5.2) and view the certificates in human-readable form (see section 6.6.5).

### 8.1.5 Generate an Individual Secret Keys

Because public key cryptography is not suited for the encryption of bulk data, Alice and Bob need to create a file with pseudo-random data that will be used as the shared secret for a symmetric encryption algorithm. In the following examples, the AES cipher with a key length of 256 bits and CBC mode will be used:

- In Alice's directory:

```
openssl rand \
     -out alice_sec.bin 16
```

- In Bob's directory:

```
openssl rand \
     -out bob_sec.bin 16
```

These shared secrets must either be encrypted for the exchange over a public channel or exchanged via a secure channel.

### 8.1.6 Exchange the Certificate

In order for Alice and Bob to exchange messages successfully, the individual certificates need to be exchanged. Execute the following commands from the base directory:

```
copy \
     Alice\alice_crt.pem Bob\
copy \
     Bob\bob_crt.pem Alice\
```

At this point, Alice and Bob are ready to communicate confidentially via a public channel.

## *8.2 Example 1*

In this example, Alice and Bob will use S/MIME for communication because it provides a format that allows important parameters, like the implemented cipher, to be encoded inside the message.

The example only demonstrates the transmission and reception of a single message from Alice to Bob. The opposite direction works analogously. Please choose a plaintext message and save it to `Alice\plaintext.txt`.

### 8.2.1 Ensure Data Integrity and Authentication

In Alice's directory, the following command signs the plaintext message with Alice's private key:

```
openssl smime \
     -sign \
     -signer alice.pem \
     -nocerts \
     -in plaintext.txt -out smimetext.txt
```

**Further Study:** Examine the S/MIME message. Embed and view Alice's certificate in the S/MIME message (see section 6.14.1). Explore the `-to`, `-from` and `-subject` parameters of the `smime` command.

### 8.2.2 Ensure Confidentiality

In this step, we encrypt the message with Bob's certificate, save it to the base directory (representing the transmission over the public channel) and decrypt it with Bob's private key. Execute the following command in the base directory:

- Encrypt and send the message:

```
openssl smime \
     -encrypt \
     -in Alice\smimetext.txt \
     -out alice_to_bob.enc \
     Alice\bob_crt.pem
```

- Receive and decrypt the message:

```
openssl smime \
     -decrypt \
     -recip Bob\bob.pem \
     -in alice_to_bob.enc \
     -out Bob\alice_to_bob.txt
```

**Further Study:** Examine the S/MIME message.

### 8.2.3 Verify Data Integrity

Now that Bob has received and decrypted the confidential message from Alice, he needs to verify that the message was, in fact, created by Alice and that the integrity of the message was not compromised:

```
openssl smime \
     -verify \
     -CApath trusts \
     -in alice_to_bob.txt
```

**Further Study:** Verify the signature against a specific certificate (see section 6.14.2).

## *8.3 Example 2*

This example does not feature a handy encoding to transmit meta data with the signed and encrypted data, like the implemented cipher and digest. Instead it demonstrates the raw commands necessary to securely exchange a shared secret, sign and encrypt data and transmit it.

This example only presents the transmission and reception of a single message from Alice to Bob. The opposite direction works analogously. Please choose a plaintext message and save it to `Alice\plaintext.txt`.

### 8.3.1 Exchange Secret Keys

Due to the fact that public key cryptography is not suited for the encryption of bulk data, Alice and Bob need to exchange their shared secrets before they are able to exchange data in an efficient manner. By using public key cryptography for the key exchange, only a small amount of data is processed and the confidentiality of the shared secrets is ensured:

- Alice's shared secret is encrypted with Bob's certificate and transmitted over the public channel by saving it to the base directory:

```
openssl rsautl \
     -encrypt \
     -certin -inkey Alice\bob_crt.pem \
     -in Alice\alice_sec.bin \
     -out alice_sec_to_bob.bin.enc
```

- Alice's shared secret is received over the public channel by reading it from the base directory and decrypted with Bob's private key:

```
openssl rsautl \
     -decrypt \
     -inkey Bob\bob_key.pem \
     -in alice_sec_to_bob.bin.enc \
     -out Bob\alice_sec.bin
```

### 8.3.2 Ensure Data Integrity and Authentication

In this step, Alice's message is hashed using the SHA1 message digest, signed with Alice's private key and transmitted over the public channel by saving it to the base directory:

```
openssl dgst \
     -sign Alice\alice.pem \
     -sha1 \
     -out alice_to_bob_signature.txt Alice\plaintext.txt
```

**Further Study:** Use different message digests (see section 3.2.2).

### 8.3.3 Ensure Confidentiality

Using Alice's shared secret, the message is encrypted with the aes-256-cbc cipher which uses the symmetric encryption algorithm AES with a key length of 256 bit in CBC mode:

```
openssl enc \
     -e -aes-256-cbc -base64 \
     -pass file:Alice\alice_sec.bin \
     -in Alice\plaintext.txt \
     -out alice_to_bob_message.enc
```

Bob needs to have prior knowledge of the cipher and the parameters used for encryption because this information is not encoded in the message. The following command demonstrates how Alice's shared secret is used to decrypt the message:

```
openssl enc \
     -d -aes-256-cbc -base64 \
     -pass file:Bob\alice_sec.bin \
     -in alice_to_bob_message.enc \
     -out Bob\alice_to_bob_message.txt
```

**Further Study:** Use different ciphers for encryption (see section 2.2) and remember to generate the appropriate encryption key (see section 8.1.5). Use different digests, specified by the `-md` parameter, to create the shared secret from the passphrase.

### 8.3.4 Verify Data Integrity

The following command demonstrates how the received signature is verified using Alice's certificate. It verifies successfully if the signature was generated using Alice's private key:

```
openssl dgst \
     -verify Bob\alice_crt.pem \
     -sha1 \
     -signature alice_to_bob_signature.txt \
     Bob\alice_to_bob_message.txt
```

Bob needs to have prior knowledge of the message digest algorithm because this information is not encoded in the message.

**Further Study:** Verify the message against a specific certificate or a collection of trusted certificates (see section 6.9.3).

# 9 Concerning „Ten Risks of PKI"

In this section, I will comment on a famous paper on the risks of Public Key Infrastructures "*Ten risks of PKI: What You're not Being Told about Public Key Infrastructure*" Carl Ellison and Bruce Schneier as well as the two reactions "*Response to Ten Risks of PKI*" by Aram Perez and "*Seven and a Half Non-risks: What You Shouldn't Be Told about PKI*" by Ben Laurie which show some risks to be non-existent. Although the first paper presents several issues that cannot be overlooked or denied, it concludes that Public Key Infrastructures should not be used and that we are better off without them.

In my opinion, it is rather short-sighted to discourage the use of PKI because it solves several issues and addresses important requirements for the exchange of data. Systems architects need to be shown solutions and the accompanying security implications. Therefore, I feel it is necessary to comment on these ten risks. After reading the previous sections about the design issues, you are now well prepared for this discussion. Public Key Infrastructures may not solve all issues in modern cryptography but reduce the number of risks which a systems architect needs to take into account in comparison to public key cryptography and symmetric cryptography.

In *Ten Risks*, the author demonstrates his short-sightedness by stating that there is no need for client certificates. Authorization often needs to go both ways. This is also true for e-commerce where the seller needs to be convinced of the identity of the customer.

## 9.1 Risk #1: Trust

The first risk which the authors present is the impact of the trust which you place into a Certificate Authority. Therefore, careful consideration is required because the Certificate Practices Statement of a CA needs to be acceptable. This was discussed in detail in section 6.16.1.

In the second part, the authors assume that a CA grants authority, i.e. the permission to perform a special task. This is utterly wrong because a Public Key Infrastructure simply enables two or more parties to be assigned roles, to authenticate and authorize each other to perform a special task.

## 9.2 Risk #2: Integrity of Private Keys

The integrity of the private key is identified by the authors to be an integral part of Public Key Infrastructures. This is discussed in section 6.16.3.

Although a compromised private key introduces a number of attacks against the involved cryptographic systems, the authors fail to mention that every cryptographic system has some kind of private component which must be secured against misuse. By omitting this fact, the authors implicitly assume that the reader is aware of this fact and make public key cryptography seem less secure than classic cryptography, i.e. symmetric cryptography. I do not assume that it was their intention to mislead the reader.

Personal computers were not designed to provide a secure compartment for critical data like a shared secret and the private components of a key pair. They are more susceptible to attacks (via a network or by interpreting the emitted electromagnetic waves) than a specialized device created for the sole purpose of storing critical data.

Consequently, a systems architect needs to be aware of those facts in order to be able to take measures against this risk. The following list shows several exposed points:

- The private key must be encrypted and protected by a strong passphrase.
- If the private key is stored on a smart card, the owner needs to report it missing or stolen as soon as possible.
- If the private key is stored on a smart card, the transmission into the processing device needs to be secured and trusted.
- When the private key is decrypted for usage, it needs to be handled in a secure manner to protect it against eavesdropping.

## 9.3 Risk #3: Integrity of the Key Store

When utilizing channels protected by public key cryptography, a common task is the verification of the presented certificates. Consequently, the collection of root certificates becomes critical data. A root certificate represents an authority that is trusted to follow its Certificate Practices Statement and, therefore, to ensure that the identity information of the owner of a private key was thoroughly checked before issuing the corresponding certificate. If the key store has been compromised, the attacker is able to import a forged root certificate to support a spoofing attack in which the user is redirected to a site that is verified correctly by the forged root certificate.

But there is another facet to consider: even if the key store remains uncompromised, will the user be able to recognize an attack? Can he understand the implications of a security warning presented by an application like a browser? Please refer to section 6.16.8 for an in-depth discussion.

## 9.4 Risk #4: Uniqueness of a Common Name

If two users apply for the same Relative Distinguished Name in a single realm[71], the conflict needs to be resolved because the Distinguished Name of a user needs to be unique. The uniqueness of a DN is required because it represents the identity information of a user and a single entity cannot have more than one identity. Consequently, roles are not supported lest they be encoded in the Common Name element of the DN.

In *Ten Risks*, the author argues that such a conflict is impossible to produce because certificates are issued for fully qualified DNS names which are unique by design, i.e. `CN=hostname`. This perception is caused by a rather limited view on the subject because, apart from server certificates, several other types are supported by the X.509 standard and its extensions (refer to section 6.2.1).

71 Refer to section 6.1 for an introduction to X.500 based directories.

## 9.5 Risk #5: Authority

The authors question the fact that a Certificate Authority is an authority and ask why it may grant the permission to utilize SSL/TLS. They must have noticed that the CA is an authority on identity verification because that is its purpose by design. It certifies that the identity of an applicant is valid and that the private key corresponding to the presented public key is in the applicant's possession at the time the request is generated.

Based on this identity verification, a CA enables the usage of certificates by providing an infrastructure with formalized processes in which the user places trust. The user may also decide not to trust a CA and, therefore, forbid secured connections to be established with entities presenting a certificate signed by such a CA.

It is argued indirectly that an uncertified server should be allowed to utilize SSL/TLS. This statement entirely misses the point because the SSL/TLS protocols are based on X.509 Public Key Infrastructures. An uncertified server is merely able to utilize symmetric cryptography to provide confidentiality but it needs to implement another method to authenticate against the user and vice versa.

## 9.6 Risk #6: Impact of the User

The authors attempt to discuss whether the user is "part of the design" but continue to criticize that SSL/TLS does not extend to the application level and, thereby, does not certify the validity of the presented content. These two issues do not overlap except when discussing whether a user is able to recognize an attack as I have done in section 6.16.8. In my opinion, even the ignorant user is able to recognize unexpected content in most cases. For the sake of completeness, let's take a look at this by sticking with the web server example:

- If the web server was taken over to serve unauthorized content, the user may be tricked into accepting it. This attack will have succeeded and the security of the private communication will have been compromised. Still, this can hardly be blamed on SSL/TLS or Public Key Infrastructures, in general, because of the complexity and impracticality of certifying the security of each and every web server.
- If the user was redirected to an unauthorized web server (i.e. a classical spoofing attack), the presented certificate may be impossible to verify correctly provided that the corresponding private key remains uncompromised. Refer to section 6.16.8 for a discussion whether the user can be expected to recognize the threat.
- For the sake of completeness, the user may have been redirected to content that differs greatly from the expected content. It was proposed by the authors, that a user could be presented with foreign content which is obviously recognized to originate from another company. In my opinion, this should be considered a serious insult because a user is able to differentiate between the content he requested and the content he is presented in such a case.

The section about this risk did, in fact, address two justified but separate issues which are intermixed. The risk was presented in an unfavourable manner and the reader is confused or even mislead.

## 9.7 Risk #7: Composite Systems

The authors present an implementation in which some tasks of the Certificate Authority are fulfilled by a *Registration Authority (RA)*. It is argued that this design is less secure than a sole CA.

Although the combination of a CA and a RA causes more elements to be involved and more complex processes to be implemented, in reality, this argument is pedantic and not necessarily true. Mathematically, there is a higher probability that any single component fails, but the authors do not show whether this probability can be perceived by the user or is negligibly small.

Similar to section 6.16.2, this argument boils down to a trust issue so that the architect needs to decide whether this design can be trusted.

## 9.8 Risk #8

Before a certificate is issued, the Certificate Authority needs to identify the applicant. The authors criticize that the identity of the applicant is established by an unknown process. If a trustworthy CA is selected, this is not true.

Firstly, this process needs to be defined in the Certificate Authority's Certificate Practices Statement. Secondly, as I have argued before, the user needs to trust the CA to follow its CPS before signing in to its service or trusting its certificate. Please refer to section 6.16.1 for details.

The authors also seem to expect that the CA needs to certify that the owner of a certificate remains in control of the corresponding private key even after the certificate was issued. In fact, the CA does not even certify that the applicant is in control of the private key at the time the certificate is enrolled.

In my opinion, the presented risk is not composed of two but only of a single issue:

- The user needs to decide whether he or she wants to trust the CA. This issue was already discussed in risk #1 (see 9.1).
- A Certificate Authority is an authority for identity verification which binds a presented identity to a public key. Therefore, it never makes any assumptions or assurances about the private key. Consequently, the certificate only proves that the applicant was able to present a public key during enrolment or that he was in control of the private key before or during enrolment.

## 9.9 Risk #9: Certificate Practices

When a Certificate Authority is founded, it creates a Certificate Practices Statement that needs to be published so that a potential customer is able to revise it. A decision for or against a CA should be based primarily on the CPS instead of the pricing model.

Although the authors have stated that several parameters need to be preset in a CPS, like key length, certificate lifetime or the implementation of revocations, the potential customer is hardly forced to choose a certain CA over another. Considering the number of commercial Certificate Authorities, a potential customer is able to match a CPS to his or her needs.

## 9.10 Risk #10: Single Sign-On

The authors close their list by stating that Single Sign-On is the "killer application" for certificates. In fact, I need to agree with the comments in the reactions to *Ten Risks* that they are not necessarily tied together and that commercial products often rely on symmetric cryptography to authenticate users.

## 9.11 Summary

Although the authors have raised some justified arguments and, thereby, exposed several risks that have to be taken into account when designing and implementing cryptographic systems based on Public Key Infrastructures, they made the mistake of not presenting the arguments in a concise and comprehensive manner. Some lines of thought are left to be finished by the reader and some of the presented issues have been intermixed with one or more other.

In my opinion, a paper needs to contain a clear set of arguments which are introduced and presented individually building a concise chain of arguments. This is especially true when arguing against the subject of the paper. In *Ten Risks*, the authors come dangerously close to what the internet community calls FUD, i.e. fear, uncertainty, doubt.

Please refer to section 6.16 for a thorough discussion of risks and issues.

# 10 Summary

This document provides an introduction to Public Key Infrastructures by explaining the concepts which are involved and by presenting two well-known implementations: X.509 PKI and OpenPGP PKI. Every task is demonstrated using the OpenSSL library, a free implementation of the Secure Socket Layer v2/v3 and Transport Layer Security v1 protocols and GNU Privacy Guard, a free implementation of the OpenPGP standard. A thorough presentation of the subject is offered by covering the fundamentals necessary to achieve the necessary knowledge of the concepts: symmetric and public key cryptography. A dedicated chapter demonstrates the importance of one-way functions in modern cryptography.

In addition to the development of stronger cryptography and more secure systems, the scalability improves in the course of this document, because the number of trust relationships that need to be established by the participating entities is reduced significantly, increasing the scalability of modern cryptographic systems. At the same time the number of entities, which are competing for users and their trust, become critical elements of the cryptographic system, introducing issues that can be misused to reduce or even compromise the security of the system. These issues are discussed thoroughly to allow a cryptographic system to be evaluated.

The document concludes by commenting on a famous paper on risks that are introduced by Public Key Cryptography.

# 11 Appendix: Benchmarking and Sizing

During the design and implementation of a product which uses cryptography to secure the exchange of messages and data, it is inherently hard to determine the maximum number of concurrent users such a system can serve. It is very important to be aware of this limit because exceeding it causes the system to become overloaded. The consequence of an overloaded system is usually a significant drop of its performance. You are well advised to prevent this to happen.

If the system was purchased, the vendor may be offering some information on the number of concurrent connections and the maximum throughput. These numbers are often presented without relating them but in a system that makes heavy use of cryptography, the performance may vary significantly depending on the number of concurrent connections and their throughput. Therefore, this data allows for some interpretation[72].

Independently of these considerations, this document does not provide a recipe to benchmark a system and size it. It rather provides some insight what typical encrypted payload consists of and how some measure for the performance of the processing system can be derived.

Fortunately, the OpenSSL library implements the `speed` command to benchmark the performance of all message digests, ciphers and public key algorithms. This allows for performance data to be generated for the executing system.

## *11.1 Dissection*

Before the system can be benchmarked, you need to identify the different components of the applied algorithms. The following list presents some scenarios in which cryptography is used to achieve different objectives:

- Authentication/Integrity

  Such traffic is usually divided into equally sized blocks which are first hashed using a message digest (see section 3.2.2) and then signed using public key cryptography (see section 4).

- Confidentiality

  This feature of cryptography can be achieved by either symmetric or public key cryptography. It is not recommended using public key cryptography to encrypt bulk data because it is slower than symmetric algorithms by several magnitudes. Therefore, the former is only used for authentication and the exchange of shared secrets.

- Authentication/Integrity and Confidentiality

72 It may even be questionable whether the performance information was deduced from a typical instead of a favourable scenario.

This objective requires a combination of the two above because usually encrypted using a symmetric algorithm and appended with a signed message digest.

It is important to have some intimate knowledge of the implemented protocols because a single user may require more than a single connection to be maintained:

- If the encrypted traffic is simply routed through the system, you do not need to take into account any cryptography. The performance can be measured by simply taking into account the throughput of the system.
- If the encrypted channel is terminated by the system, the load caused by the corresponding user depends on how the data is handled:
  - Is it processed locally and answered through the same channel?
  - Is it routed to another system? Is the outgoing traffic encrypted causing a second connection to be maintained?
- A client may also require more than one connection to be maintained to perform related tasks like a control and data connection. This places a proportionally higher load on the system.

There are obviously several facts to be taken into consideration to perform a thorough performance analysis and there is still another aspect to consider: It is important to identity the amount of data that needs to be hashed or encrypted in a certain interval of time. The commands presented in the following sections provide results as a number of bytes or operations that were processed per second.

Last but not least, the results of the presented commands may differ from the actual performance data because the parameters which were used to compile the OpenSSL library may cause a significant difference in the performance of the test cases.

## *11.2 User Behaviour*

Before being able to benchmark a system you need to define the typical behaviour of a user to model the load. A user might typically cause a certain throughput or number of sign and verify operations per second.

**Note:** It is usually rather hard or even impossible to define the behaviour of a typical user because the performed tasks may cause a greatly varying load on the system.

## *11.3 Message Digests*

All of the digests supported by the OpenSSL library can be passed to the speed command as parameters to benchmark the performance of the executing system: `md2`, `md4`, `md5`, `hmac`, `sha1`, `sha256`, `sha512`, and `rmd160`.

```
openssl speed \
     DIGEST
```

**Warning:** The name of the cipher of cipher suite is case sensitive.

## 11.4 Symmetric Encryption

Due to the fact that the OpenSSL library supports a variety of ciphers (see section 2.2) they can either be measured as a cipher suite or, if applicable, with parameters specifying the mode and key length. The following list of cipher suites is supported: `idea`, `rc2`, `des`, `aes`, and `blowfish`.

If the possible parameters are taken into account, the list of ciphers becomes rather lengthy: `idea-cbc`, `rc2-cbc`, `bf-cbc`, `des-cbc`, `des-ede3`, `aes-128-cbc`, `aes-192-cbc`, `aes-256-cbc`, `rc4`.

```
openssl speed \
     CIPHER
```

**Warning:** The name of the cipher of cipher suite is case sensitive.

## 11.5 Public Key Cryptography

Contrary to the other parameters of the speed command, the public key algorithms do not offer their results in the amount of data that was processed in a certain interval of time but rather in operations, i.e. sign and verify, per second.

The parameter is composed of the public key algorithm followed by the required key length: `rsa512`, `rsa1024`, `rsa2048`, `rsa4096`, `dsa512`, `dsa1024`, and `dsa2048`.

```
openssl speed TYPE
```

**Note:** The RSA public key algorithm can also be specified without a key length, i.e. `rsa`, to allow all key lengths to be compared after a single run.

**Warning:** The name of the desired public key algorithm is case sensitive.

## 11.6 SSL/TLS Connections

In addition to the performance analysis of individual algorithms with the speed command, the OpenSSL library also allows SSL/TLS connections to be benchmarked in their entirety. The following commands require the name and port of a host which is to be analyzed:

- Measure SSL handshakes using a new session ID for every connection:

  ```
  openssl s_time -connect HOST:PORT -new
  ```

- Measure SSL handshakes using the same session ID for every connection:

  ```
  openssl s_time -connect HOST:PORT -reuse
  ```

- Measure SSL handshakes and transfer a small amount of payload by requesting the page at `/`:

  ```
  openssl s_time -connect HOST:PORT -www /
  ```

- Measure SSL handshakes for `N` seconds:

```
openssl s_time -connect HOST:PORT -time N
```

**Note:** By omitting both the `-new` and the `-reuse` parameters, both variants are executed to produce the performance results.

**Further Study:** Similar to the `s_client` command (see section 6.15), the `s_time` command can be configured to match the required certificate verification using the `-CAfile`, `-CApath`, and `-verify` parameters.

**Further Study:** Like the `s_client` command (see section 6.15), the `s_time` command allows client certificates to be taken in to account using the `-key` and `-cert` parameters.

**Further Study:** To benchmark unencrypted connections, add the `-cipher NULL-SHA` parameter. For more information how to control the supported cipher suites, refer to section 6.15.4.

## *11.7 Interpretation*

After the dissection and the analyses of the expected traffic were covered in the previous sections, the results can be used to deduce the overall performance of the system which is commonly measured in concurrently served users.

You need to identify the component with the least performance, e.g. due to limited processing power or due to limited bandwidth. For this component you are required to choose the appropriate metric: the throughput measured in bytes per second describes the load of bulk encryption or sent and received data whereas a number of operations per second needs to be used for public key operations like signing and verifying. Consequently, a number of concurrently served users can be calculated with a maximum throughput.

It is rather important to preserve the relation between three pieces of data: Performance data cannot be applied to a system which is based on different hardware. If the behaviour of the user changed over time, the benchmarking process needs to be repeated. Changes in the design or implementation of the cryptographic system may cause performance data to become outdated.

# 12 Appendix: Resolving Error Codes

If the OpenSSL library encounters an error, it provides the user with a more or less meaningful message. It usually contains a cryptic hint what has caused the error.

The `errstr` command is meant to provide a more helpful message. It expects a hexadecimal code as its parameter which can be found after the second colon in an error message of the OpenSSL library.

Unfortunately, the default error message often provides the only helpful hint:

- If the verify command is used to verify a Certificate Signing Request instead of a certificate, it displays the following error message:

```
unable to load certificate

5012:error:0906D06C:PEM routines:PEM_read_bio:\
     no start line:.\crypto\pem\pem_lib.c:644:
     Expecting: TRUSTED CERTIFICATE
```

The verification failed because the verify command was expecting a file starting with `TRUSTED CERTIFICATE` but instead got `CERTIFICATE REQUEST`. For someone familiar with the contents of files containing PEM-encoded objects, the source of the error is obvious.

- The `errstr` command is expected to provide a more helpful command. After extracting the error code `0906D06C` from the above message and feeding it to `errstr`, it offers the following message:

```
error:0906D06C:PEM routines:PEM_read_bio:no start line
```

The information obtained from `errstr` was already encoded in the original error message and does not provide any further insight as to the source of the error.

The above example demonstrated that `errstr` cannot be expected to offer the Ultimate Hint™ but, rather, should be remembered as a tool which may be of help.

# 13 Appendix: Microsoft's “makecert”

The command line tool `makecert.exe` is provided my Microsoft to support creating X.509-based certificates for testing purposes. It is included in the Platform SDK and the .NET Framework SDK.

The following sections provide an introduction to this tool and how to perform common tasks on and with certificates.

**Warning:** It is assumed that you are familiar with certificate stores in Windows-based operating systems.

## *13.1 Creating a certificate*

When creating a new certificate using makecert, it inherits the following default values:

- Validity: Until the end of 2039 (2039-12-31 23:59:59)
- Key length: 1024 bit
- Hash algorith: MD5

See section 13.3 for details how to change the default values.

### 13.1.1 Saving to a File

A new certificate may be saved to a set of files consisting of a private key (`.pvk`) and a certificate (`.cer`). The certificate is saved DER-formatted and can be view by using the commands introduced in section 6.6.5.

The following command creates a new certificate which is saved to a certificate file accompanied by a private key file:

```
makecert -sv cert.pvk cert.cer
```

### 13.1.2 Saving to a Certificate Store

Alternatively, certificates can be saved to the certificate store and private key to a container.

The following command saves a newly creates certificate to the certificate store `testCertStore` and the corresponding private key to the key container named `testKeyContainer`.

```
makecert -sk testKeyContainer -ss testCertStore
```

**Note:** Well-known names of certificate stores include `My` (Personal Certificates).

**Further Study:** By using the parameter `-pe`, a private key can be marked exportable. Otherwise, the private key is bound to a single machine.

In addition, it is possible to determine in which context a certificate is saved to a certificate store. This can either be in the store of the current user (`currentuser`) which is the default or in the store of the local machine (`localmachine`):

```
makecert -ss testCertStore -sr localmachine
```

### 13.1.3 Specifying the Subject

The previous commands have caused a newly created certificate to be issued to a default subject name. The following command demonstrates how to customize the subject's distinguished name:

```
makecert -n „CN=Test‟ -sv cert.pvk cert.cer
```

## *13.2 Creating a Self-Signed Certificate*

As introduced in section 6.4 self-signed certificates are usually used for testing purposes when having a Certificate Authority issue a new certificate is too expensive or too time consuming. The following command creates such a self-signed certificate:

```
makecert -r -n „CN=Test‟ -cy end -sv RootCA.pvk RootCA.cer
```

Due to the fact that root CAs are based on self-signed certificates, the following command is rather similar to the previous one and allows more certificates to be issued:

```
makecert -r -n „CN=Test‟ -cy authority -sv RootCA.pvk RootCA.cer
```

**Further Study:** See section 6.6 for a detailed discussion of Certificate Authorities.

## *13.3 Specifying Properties of a Certificate*

As mentioned before, a new certificate inherits default values for most of its properties. The following parameters allow all of these properties to be specified.

- Key length:

  ```
  makecert -len 2048
  ```

- Hashing algorithm (MD5, SHA1):

  ```
  makecert -a sha1
  ```

  **Further Study:** See section 3.2 for an introduction to hashing algorithms and details abount MD5 and SHA1.

- Start of the validity period:

  ```
  makecert -b 12/31/2007
  ```

- End of the validity period:

  ```
  makecert -e 12/31/2007
  ```

- Duration of the validity period in months:

```
makecert -m 5
```

**Example:** The following command creates a certificate:

```
makecert -b 01/01/2007 -m 12 -sv Test.pvk Test.cer
```

### *13.4 Creating and Signing a certificate*

While the previous commands created a certificates issues by „Root Agency", this section introduces command line switches to specify a custom issuer to sign a new certificate.

**Note:** By using the parameter `-cy authority`, a intermedia CA is created.

#### 13.4.1 Using a File-Based Issuer

The following command demonstrates how to create a new certificate which is signed by another certificate's private key which is specified by file:

```
makecert \
     -ic RootCA.cer -iv RootCA.pvk \
     -n "CN=Nic" \
     -cy end
     -sv Nic.pvk Nic.cer
```

#### 13.4.2 Using a Store-Based Issuer

Similar to the command in the previous section, the following command creates a new certificate which is signed by another certificate's private key which is taken from the specified certificate store:

```
makecert \
     -is CurrentUser -ir My \
     -n "CN=My Root CA" \
     -cy end \
     -sv Nic.pvk Nic.cer
```

**Note 1:** The parameters -is and -ir are equivalent to -ss ad -sr introduced in section 13.1.2.

**Note 2:** It is essential to specify the issuer's distinguished name when using a certificate from a certificate store because the system needs to know which certificate to use.

# 14 Appendix: Mozilla's "certutil"

`certutil` is part of the Network Security Sevices (NSS)[73] which is one of the PKI projects by the Mozilla Project[74]. It is based on the Netscape Portable Runtime (NSPR)[75] and support SSL (v2 and v3), TLS, PKCS#5, PKCS#7, PKCS#11, PKCS#12, S/MIME and X.509-based certificates.

This chapter introduces how certutil works and how common tasks are performed.

## *14.1 Introduction*

Contrary to OpenSSL which leaves the handling of certificates to the user and only allows for certificate tasks to be performed, certutil provides the user with a collection of databases which can be used to store pieces of information:

- `cert8.db`: The certificate database stores all certificates.
- `key3.db`: If a certificate is accompanied by a private key, it is stored in the key database.
- `secmod.db`: Users can decide what kind of trust to place into a certificate. This information is stored in the trust database.

Although the following section provide a thorough introduction to certutil, its help is detailed. It is accessed by the following command:

```
certutil -H
```

**Note:** By issuing `certutil -h`, a compressed version of the help is displayed.

### 14.1.1 Creating the Database Files

The following command creates the database files introduced in the previous section in the current directory:

```
certutil -d . -N
```

**Further Study:** It is possible to manage several sets of database files in the same directory. To tell them appart, the parameter `-P dbprefix` allows a string to be specified which is prefixed to the name of the database file.

### 14.1.2 Trust Attributes

Users can place trust in a certificate in three categories: SSL, emailing and object signing. The individual trust attributes for each of the categories are comma-separated. For a list ofsupported trust attributes, refer to Table 6.

73 http://www.mozilla.org/projects/security/pki/nss/
74 http://www.mozilla.org
75 http://www.mozilla.org/projects/nspr/

```
... -t TCu,TCu,TCu ...
```

| ***Trust Attribute*** | ***Description*** |
|---|---|
| p | Valid peer |
| P | Trusted peer (implies p) |
| c | Valid CA |
| T | Trusted CA to issue client certificates (implies c) |
| C | Trusted CA to issue SSL server certificates (implies c) |
| u | User certificate |
| w | Send warning |
| g | Make step-up certificate |

*Table 6: Trust Attributes for certutil*

## 14.2 Listing Keys and Certificates in the Database

The following command lists all private keys in the key database:

```
certutil -d . -K
```

**Further Study:** This command supports parameters `-k` and `-f` as explained in section 14.3.3.

The following command allows certificates residing in the database to be listed and explored:

```
certutil -d . -L
```

**Further Study:** Display a specific certificate by using the `-n cert-name` parameter. The output will be as `openssl x509 -text -noout` (see section XXX for details).

The specified certificate can also be retrieved DER-formatted by using `-r` and PEM-formatted by using `-a`.

In addition to viewing individual certificates, the following command allows certificate chains to be displayed:

```
certutil -d . -O -n "Test Cert"
```

## 14.3 Quickstart

This section describes the commands to quickly start generating certificates for testing purposes.

### 14.3.1 Creating a Self-Signed CA Certificate

The following command creates a new self-signed certificate for a root Certificate Authority. A certificate named „Test CA“ is created with a subject name of „CN=Test“ which may be used to issue new client and server certificates.

```
certutil -d . -S -n "Test CA" -s "CN=Test" -x -t T,C
```

**Note 1:** The parameter `-x` causes the certificate to be self-signed.

**Note 2:** If a simple server certificate is quired, refer to section 14.1.2 to learn how to modify the parameter `-t`.

**Note 3:** More useful parameters are introduced in section 14.3.3.

### 14.3.2 Creating a New Certificate

After creating a certificate for a root Certificate Authority, it an be used to issue more certificates by using the following command. It creates a certificate named „Test Cert“ with the subject name „CN=Test2“ which is issued by the CA named „Test CA“.

```
certutil -d . -S -n "Test Cert" -s "CN=Test2" -c "Test CA" -t u
```

**Note 1:** To create an end-user certificate, the parameter `-x` is substituted by `-c issuer-name`.

**Note 2:** This command accepts the same parameters as described in the previous section.

### 14.3.3 Specifying Common Properties

The previously introduced commands accepts a collection of common parameters to configure their behaviour. These include the following:

- When the private key is created, the key type and size is adjusted by using the following parameters:
  - Key type: `-k key-type`

    Valid values are `dsa`, `rsa` and `ec`. It defaults to `rsa`.
  - Key size: `-g key-size`

    Valid values range from 512 to 8192. It defaults to 1024.
- The validity of a certificate is specified by `-v valid-months` which defauls to 3.
- The start of a certificate's validity can be set in the future by specifying `-w warp-months` in which `warp-months` can also be nagative to have the validity period start in the past.
- The subject's phone number is provided by `-p phone-number`.

- By default, a certificate receives a randomly generated serial number. This behaviour is overridden by `-m serial-number`.

**Further Study 1:** If a command is required to operate on the key database, it asks for a password interactively. This can be overridden by `-f password-file`.

**Further Study 2:** The commands allow for several extensions to be added to a certificate[76] which can be specified by the following parameters. Note that some or all of them operate interactively.

Add the key usage extension by using `-1`. The user is prompted to select the required usage of the certificate and private key.

Add the basic constraint extension by using `-2`. The user is prompted to select the required certificate constraint.

Add the authority keyID extension by using `-3`. The user is prompted to select an authority keyID to distinguish several certificates with the sam esubject name.

Add a CRL distribution point by using `-4`. The user is prompted to enter the URL of such an distribution point.

Add a Netscape certificate extension by using `-5`.

Add an extended key usage extension by using `-6`.

Add a comma-separated list of email addresses to the subject alternative name by using `-7 email-addresses`.

Add a comma-separated list of DNS named to the subject alternative name by using `-8 dns-names`.

## *14.4 Validating a Certificate in the Database*

The followingcommand is used to validate a certificate that is contained in the database:

```
certutil -d . -V -n „Test Cert“ -u V
```

To validate a certificate, a usage must be specified according to the values described in Table 7.

76 Extensions of a X.509-based certificate are explained unter http://developer.netscape.com/docs/manuals/cms/41/dep_gide/ext.htm

| ***Certificate Usage*** | ***Description*** |
|---|---|
| C | SSL client |
| V | SSL server |
| S | Email signer |
| R | Email recipient |
| O | OCSP status responder |

*Table 7: Certificate Usage for Validation with certutil*

Further Study 1: By specifying the `-e` parameter, the signature is checked.

Further Study 2: When validating a certificate, it is possible to assure its validity for a certain amount of time in the future by using the parameter `-b YYMMDDHHMMSS[+HHMM|-HHMM|Z]`.

## *14.5 Creating a Certificate Request*

The following command creates a new certificate request by generating a new key pair, saving in in the key database and using the supplied subject name (`-s`) to create the request. The request is printed DER-formatted to standard output.

```
certutil -d . -R -s „CN=Test‟ -o request.cer
```

**Note 1:** The request is not saved to any one of the database files.

**Note 2:** A PEM-formatted certificate can be obtained by specifying the `-a` parameter.

**Further Study:** This command accepts parameters `-k`, `-g`, `-f`, `-h` and `-p` as explained in section 14.3.3.

## *14.6 Creating a Certificate from a Request*

After creating a certificate request in the previous section, the following command demonstrated how to obtain a certificate from a request. The command specifies the issuer with the parameter `-c`.

```
certutil -d . -C -c "Test CA" -i request.der -o cert.der
```

**Note 1:** The newly created certificate is not added to any the databases.

**Note 2:** It is also possible to create a self-signed certificate from a request by using the `-x` parameter instead of `-c issuer-name`.

Note 3: Omitting the parameter `-o` prints the certificate to standard output.

**Further Study:** This command accepts the parameters `-m`, `-v` and `-f` as explained in section 14.3.3. It also supports `-{1,2,3,4,5,6,7,8}`.

## 14.7 Importing a Certificate into the Database

The following command imports a certificate into the certificate database which is named „Test 3‟ and tagged with the specified trust attributes in the process:

```
certutil -d . -A -n "Test 3" -t u -i cert.der
```

**Note 1:** The certificate is expected to be supplied DER-formatted. By specifying the parameter `-a`, the command also accepts a PEM-formatted certificate.

**Note 2:** By omitting the parameter `-i`, the command expects the certificate to be supplied on standard input.

## 14.8 Deleting a Certificate from the Database

The following command deletes the certificate named „Test Cert‟ from the certificate database:

```
certutil -d . -D -n „Test Cert‟
```

## 14.9 Modifying the Trust Attributes of a Certificate

The following command modifies the trust attributes accompanying the certificate named „Test Cert‟:

```
certutil -d . -M -n „Test Cert‟ -t u
```

**Note:** Valid values for the parameter -t are explained in section 14.1.2.

## 14.10 Create a New Key Pair

The following command generates a new key pair which is imported to the key database:

```
certutil -d . -G
```

**Further Study 1:** The command accepts parameters `-k`, `-g` and `-f` as explained in section 14.3.3.

**Further Study 2:** When a pseudo-random number is generated, a seed can be specified by using the parameter `-z noise-file`. This allows hardware random number generators to be used. The minimum file size is 20 bytes.

## 14.11 Further Study

This section contains various topics which the motivated reader is encouraged to endulge upon.

The parameter `-h token-name` allows for hardware devices to be used. There are also two internal slots which are reserved for key and certificate services (slot number 2 named `internal`) and cryptographic services (slot number 1).

When generating RSA private keys, the key's exponent can e specified by the parameter `-y exponent`. Valid values are 3, 17 and 65537.

When generating DSA keys, its PQG value can be specified by using the parameter `-q value`.

When generating EC keys, the name of the desired curve can be specified by using the paramter `-q curve-name`. A list of valid curve names can be obtained from the detailed help.

To allow for batch processing of commands from a file named `batch-file`, the following command can be used:

```
certutil -d . -B -i batch-file
```

Similar to importing a certificate in section 14.7, an email certificate can be imported to the certificate database by the following command. Details can be obtained from the detailed help.

```
certutil -d . -E ...
```

## Bibliography / References

The references presented herein have either been used during the research for this document or are known to contain useful information for the motivated reader.

[1] Geheime Botschaften (German); Simon Singh; Deutscher Taschenbuch Verlag (dtv); 2001; Originally published as "The Code Book. The Science of Secrecy from Ancient Egypt to Quantum Cryptography"

[2] Public Key Infrastructure Overview; Joel Weise; SunPS Global Security Practice; Sub BluePrints OnLine; August 2001

[3] An Introduction to OpenSSL, Part One: Cryptographic Functions; Holt Sorenson; Security Focus; http://www.securityfocus.com; 2001

[4] An Introduction to OpenSSL, Part Two: Cryptographic Functions Continued; Holt Sorenson; Security Focus; http://www.securityfocus.com; 2001

[5] An Introduction to OpenSSL, Part Three: PKI – Public Key Infrastructure; Holt Sorenson; Security Focus; http://www.securityfocus.com; 2001

[6] An Introduction to OpenSSL, Part Four: The SSL and TLS Protocols; Holt Sorenson; Security Focus; http://www.securityfocus.com; 2001

[7] SSL – Rumours and Reality: A Practical Perspective on the Value of SSL for Protecting Web Servers; Charl van der Walt; Security Focus; http://www.securityfocus.com

[8] Ten risks of PKI: What You're not Being Told about Public Key Infrastructure; Carl Ellison, Bruce Schneier; Computer Security Journal; November 2000; http://www.schneier.com/paper-pki.html

[9] Response to "Ten Risks of PKI"; Aram Perez; http://homepage.mac.com/aramperez/responsetenrisks.html

[10] Seven and a Half Non-risks: What You Shouldn't Be Told about PKI; Ben Laurie; http://www.apache-ssl.org/7.5things.txt

[11] Conventional Public Key Infrastructure: An Artefact Ill-Fitted to the Needs of the Information Society; Roger Clarke; http://www.anu.edu.au/people/Roger.Clarke/II/PKIMisFit.html

[12] We Are All Security Consumers; Bruce Schneier; IEEE Security & Privacy, Vol. 1, No. 1, Jan./Feb. 03; http://www.schneier.com/essay-010.html

[13] Risks of Relying on Cryptography; Bruce Schneier; Inside Risks 112, Commiunications of the ACM, vol 42, n 10, Oct. 1999; http://www.schneier.com/essay-021.html

[14] Risks of PKI: Secure E-Mail; Bruce Schneier; Inside Risks 115, Communications of the ACM, vol 43, n 1, Jan. 2000; http://www.schneier.com/essay-022.html

[15] Risks of PKI: Electronic Commerce; Carl Ellison, Bruce Schneier; Inside Risks 116, Communications of the ACM, vol 43, n 2, Feb. 2000; http://www.schneier.com/essay-023.html

[16] Security Pitfalls in Cryptography; Bruce Schneier; http://www.schneier.com/essay-028.html

[17] Customers, Passwords, and Web Sites; Bruce Schneier; IEEE Security & Privacy, July/August 2004; http://www.schneier.com/essay-048.html

[18] Cryptoanalysis of MD5 and SHA1: Time for a New Standard; Bruce Schneier; Computerworld; August 19, 2004; http://www.schneier.com/essay-074.html

[19] Authentication and Expiration; Bruce Schneier; IEEE Security & Privacy, January/February 2005; http://www.schneier.com/essay-079.html

[20] The Handbook of Applied Cryptography; Alfred J. Menezed, Paul C. van Oorschot, Scott A. Vanstone; http://www.cacr.math.uwaterloo.ca/hac/

[21] An Illustrated Guide to Cryptographic Hashes; Steve Friedl; http://www.unixwiz.net/techtips/iguide-crypto-hashes.html

[22] Moderne Verfahren der Kryptographie: Von RSA zu Zero-Knowledge (German); Albrecht Beutelspacher, Jörg Schwenk, Klaus-Dieter Wolfenstetter; 5. Auflage; Vieweg

[23] Introduction to Cryptography; Network Associates, Phil Zimmermann; 1990-1999

[24] OpenSSL Project; http://www.openssl.org

[25] OpenSSL Binary Distributions; http://www.openssl.org/related/binaries.html

[26] OpenSSL Documentation; http://www.sial.org/howto/openssl/

[27] OpenSSL Command-Line HOWTO; http://www.madboa.com/geek/openssl/

[28] Wikipedia; http://wikipedia.org

[29] CAcert; http://www.cacert.org

[30] The GNU Privacy Handbook; The Free Software Foundation; http://www.gnupg.org/gph/en/manual.html

[31] A Practical Introduction to GNU Privacy Guard in Windows; Brendan Kidwell; Brendan@glump.net; http://www.glump.net/dokuwiki/gpg/gpg_intro

[32] Krypto-Kampagne (German); Heise Zeitschriftenverlag; http://www.heise.de/security/dienste/pgp/

[33] rakshas' scrapbook; http://rakshas.de (now: http://dille.name)

[34] SimpleCert; Nicholas Dille, nicholas@dille.name; http://rakshas.de/soft/simplecert.html

[35] Digitale Signaturen: Modelle zur Gültigkeitsprüfung von Zertifikaten; Luigi Lo Iacono, Sibylle Müller, Michael Schneider; iX, Magazin für professionelle Informationstechnik; heise Verlag; Juni 2006

[36] Network Security Sevices (NSS); Mozilla Project; http://www.mozilla.org/projects/security/pki/nss/

## Version History

| ***Date*** | ***Description*** |
|---|---|
| 31.05.2007 | Added introduction to makecert.exe from Microsoft's .NET Framework SDK<br>Added introduction to certutil from Mozilla's Network Security Services (NSS)<br>Added index of tables and illustrations |
| 08.10.2006 | Reorganized several chapters to group information concerning certificate validation<br>Changed home page of the document<br>Increased version to 1.2.0 |
| 15.05.2006 | Added new item to roadmap<br>Added two sections to the introduction: Versioning, Language<br>Added a section about verification models<br>Increased version to 1.1.0 |
| 06.04.2006 | Finished section about the SSL/TLS handshake<br>Increased version to 1.0.0 |
| 03.04.2006 | Added corrections from proof-reading<br>Reviewed the appendix about benchmarking |
| 09.03.2006 | Extended the section about SSL/TLS with DH<br>Added section about SSL/TLS Handshakes |
| 06.03.2006 | Added section about PKCS#7<br>Extended the section about S/MIME with `-certfile` parameter<br>Extended the section about the creation of a CA (`index.attr` file) |
| 05.03.2006 | Added reference to Phil Zimmermann's „Introduction to Cryptography" |
| 26.02.2006 | Ported document to OpenOffice |
| 25.02.2006 | Extended documentation of `subjectAltName` |
| 24.02.2006 | Added corrections from proof-reading |
| 22.02.2006 | Added corrections from proof-reading |
| 16.02.2006 | Added some parameters to the `smime` command<br>Added several lines of text to the `verify` command<br>Added additional material to the `enc` command<br>Added an appendix about the `errstr` command |

| Date | Description |
|---|---|
| 13.02.2006 | Added corrections from proof-reading |
| 10.02.2006 | Documented parameters for revocation<br>Added cipher lists |
| 02.02.2006 | Corrected some errors in spelling and expression<br>Added some item for further study to viewing certificates and CRLs |
| 27.01.2006 | Added appendix about benchmarking and sizing |
| 21.01.2006 | Added public collections of trusted root certificates<br>Included additional material learned from "*Moderne Verfahren der Kryptographie*" in the course of the last weeks |
| 30.12.2005 | Reorganized the document to provide an general introduction to PKI followed by two practical presentations of special implementations |
| 29.12.2005 | Added several figures to present concepts visually |
| 24.12.2005 | Added a roadmap to present planned future development |
| 20.12.2005 | Added appendix about GnuPG |
| 17.12.2005 | Added a lot of content in the course of the last two months |
| 11.10.2005 | Initial structure |

## Roadmap

| No. | Description | Inclusion | Status |
|---|---|---|---|
| 1 | Verification process:<br>• build certificate chain from leaf to root<br>  • matching key identifiers, DN and serial number of a certificate's issuer and a CA's subject<br>  • the certificate of a root CA must always be trusted<br>  • appropriate certified key usage<br>• check of purpose<br>• check of trust settings<br>• check of validity | *Major* | *Open* |
| 2 | Can a special purpose of a certificate be trusted? Explore the following parameters of openssl x509:<br>• `-trustout`<br>• `-purpose`<br>• `-clrtrust`, `-clrreject` | *Major* | *Open* |

| No. | Description | Inclusion | Status |
|---|---|---|---|
| | • `-addtrust`, `-addreject`<br>• `-alias`, `-set_alias` | | |