# TLS Mastery

**Notes on TLS**

Patrick Bucher

## Contents

These notes are based on TLS Mastery by Michael W. Lucas. I highly recommend to buy that book. The examples have been modified as needed. Appendices providing instructions on how to setup a basic web and DNS server have been added.

# Chapter 0: Introduction

OpenSSL commands have the following syntax:

```
$ openssl [subcommand] [flags]
```

The flags use single dashes with long names: `-foo`, not `-f` or `--foo`.

Common flags:

- `-in`: define input (key file or the like)
- `-out`: define output
- `-text`: use textual rather than binary output

### TLS Client

The `s_client` subcommand provides a TLS-aware netcat. It can be used to fetch and output a certificate from a remote website:

```
$ openssl s_client -showcerts -connect paedubucher.ch:443 </dev/null | \
  openssl x509 -text -noout
```

There are two `openssl` commands:

1. Fetching the certificate using `s_client`:
    - `-showcerts`: show the TLS certificate
    - `-connect`: specify a `host:port` to connect to (`paedubucher.ch` on TLS port `443`)
    - `</dev/null`: do not provide any input, which is usually required from `openssl` commands
2. Parsing and displaying the certificate using `x509` (deals with X.509 certificates):
    - `-text`: output human-readable text instead of the binary representation
    - `-noout`: do not output the encoded certificate

Man pages are usually to be found with `openssl-[subcommand]`. Check `apropos openssl`.

### Regulation

The FIPS (Federal Information Processing Standards) regulates which TLS algorithms can be used. For organizations operating under FIPS regulation, those guidelines are mandatory, even though the FIPS lacks a bit behind (e.g. `SHA-1` is still considered safe).

TLS is used for TCP, DTLS for UDP protocols; they work mostly the same.

## Chapter 1: TLS Cryptography

RSA (Rivest, Shamir Adelman) and ECDSA (Elliptic Curve Digital Signature Algorithm) are the most important algorithms for private/public key pairs.

TLS uses public key cryptography to negotiate a temporary, symmetric key that is actually used to encrypt the data being transferred.

A public key infrastructure (PKI) encompasses the entire system providing cryptography (e.g. TLS, PGP).

(H)MAC: (Hashed) Message Authentication Code is a symmetrically encrypted hash (e.g. HMAC-256). A message is first hashed, then the hash is encrypted using the private key. The encrypted hash then can be decrypted against the public key.

See keylength.com for recommendations concerning secure key lengths.

## Ciphers

A cipher suite is a combination of asymmetric, symmetric, and checksum algorithms and parameters for end-to-end communication (but unrelated to the signature algorithm).

TLS 1.2 indicates cipher suites as follows: `TLS_[Kx]_[Au]_WITH_[Enc]_[MAC]`:

- `Kx`: key exchange method (e.g. ECDHE, RSA)
- `Au`: authentication method (e.g. ECDSA, RSA)
- `Enc`: symmetric encryption and mode of operation (e.g. AES, CBC, CCM, GCM)
- `MAC`: message authentication code (e.g. SHA, SHA256, SHA384)

If `Kx` and `Au` are the same, the indication is only listed once.

TLS 1.3 indicates cipher suites using a shorter form: `TLS_[Enc]_[MAC]` (`Kx`, `Au`, and `WITH` are omitted). No public key algorithm is indicated, because it is negotiated between client and server.

Different implementations use different syntaxes, use ciphersuite.info to check.

Use the `ciphers` subcommand to list supported cipher suites:

```
$ openssl ciphers -v -stdname -s -tls1_3
```

- `-v`: list one cipher per line
- `-V`: display hex values (official "names")
- `-stdname`: display standard names
- `-s`: only display supported ciphers

Example (`cut` off all but the first column):

```
$ openssl ciphers -stdname -s -v -tls1_3 | cut -f1 | sort
TLS_AES_128_GCM_SHA256
TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256
```

Ciphers are grouped in cipher lists, see `openssl-ciphers(1)` for details (e.g. `HIGH`, `MEDIUM`, `RSA`, `ECDHE` etc.). They can be listed using the `ciphers` subcommand:

```
$ openssl ciphers HIGH
```

Applications can be configured to use specific cipher lists. Sticking to `HIGH` is a good idea in general. Check ssl-config.mozilla.org for application specific configurations.

### Security Model

There are two models of trust for publik key cryptography:

1. Web of Trust: the user decides whom to trust, used for PGP.
2. Certificate Authority: audited organizations are considered trustworthy, used for TLS.

Private keys, which are enough to pretend being its identity, should only be readable/writable by the `root` user (`chmod 0600`). All certificates based on a private key have to be revoked immediately if that private key has been leaked.

Protecting private key with passphrases is usually not viable, because the passphrase needs to be entered as a service using TLS is started.

TLS resumption is a mechanism to speed up subsequent communication after TLS validation has been performed with the first request. TLS 1.2 uses session tickets (client) and server caches (server). TLS 1.3 uses pre-shared keys (PKS) and (restricted) session tickets. Resumption could be a privacy threat, because browsers can be identified by pre-shared keys and session tickets. Therefore, TLS resumption is deactivated where privacy is of high concern.

Secure renegotiation is the idea to use different levels of encryption within the same context (e.g. a web application with higher encryption for the login process than for just browsing). It was discarded, hence is missing in TLS 1.3, due to security flaws.

DHE and ECDHE (based on RSA and ECD, respectively) do not use the private key to negotiate a symmetric key used for the actual data transfer (PFS: perfect forward secrecy). Therefore, captured encrypted packets can't be decrypted using a leaked private key.

Since multiple web sites can be hosted under the same IP address, the first request must indicate a domain, so that the right TLS certificate can be picked for it (SNI: server name indication). This is done in cleartext.

## Chapter 2: TLS Connections

The `openssl s_client` subcommand is useful for debugging daemons that offer TLS-encrypted connections. Different implementations of Netcat (`nc(1)`) can deal with TLS more or less well, so better stick to `openssl`.

`s_client` was made for debugging and, therefore, also accepts invalid certificates:

```
$ openssl s_client -connect expired.badssl.com:443 </dev/null >/dev/null
$ echo $?
0
```

Specify `-verify_return_error` to fail if the certificate offered is invalid:

```
$ openssl s_client -verify_return_error -connect expired.badssl.com:443 \
  </dev/null >/dev/null
$ echo $?
1
```

### Interactive TLS Sessions

Some protocols, like HTTP, require CR+LF (carriage return and line feed: `\r\n`) to end commands, while pressing `[Enter]` on Unix terminals only sends the new line character `\n`. Add the `-crlf` option to translate a line feed into CR+LF:

```
$ openssl s_client -connect paedubucher.ch:443 -crlf
GET / HTTP/1.1
Host: paedubucher.ch
```

Press `[Enter]` twice to terminate HTTP commands; the `index.html` page will be listed.

If server name indication (SNI) is used, specify the server name so using the `-servername` flag.

Servers that offer *opportunistic TLS* (STARTTLS) allow the client to connect without TLS first and then allow the client to switch to a TLS-encrypted connection, if it wants so. The `-starttls [protocol]` can be defined to indicate that the switch to TLS is desired for the given protocol:

```
$ openssl s_client -connect mail.company.com:25 -starttls smtp
```

### Helpful Commands and Flags

Various commands can be used within an interactive TLS-encrypted session:

- `Q`: quit (cleanly close the connection)
- `k`: update the key
- `K`: update the key and request a new key
- `R`: re-negotiate the terms of the connection

Use the flag `-ign_eof` to keep the connection alife after `EOF` was sent. This also deactivates the commands above.

To only display a summary of the negotiated TLS characteristics, use the `-brief` flag:

```
$ openssl s_client -connect paedubucher.ch:443 -brief </dev/null
CONNECTION ESTABLISHED
Protocol version: TLSv1.2
Ciphersuite: ECDHE-RSA-AES256-GCM-SHA384
Peer certificate: CN = paedubucher.ch
Hash used: SHA256
Signature type: RSA-PSS
Verification: OK
Supported Elliptic Curve Point Formats: uncompressed:ansiX962_compressed_...
Server Temp Key: X25519, 253 bits
DONE
```

To only display a summary of the certificate chain, use the `-quiet` flag:

```
$ openssl s_client -connect paedubucher.ch:443 -quiet </dev/null
depth=2 O = Digital Signature Trust Co., CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = R3
verify return:1
depth=0 CN = paedubucher.ch
verify return:1
```

### Constraining TLS Versions and Ciphers

By default, `s_client` uses the highest version of TLS offered. The protocol version can be specified using the flags `-tls1_3`, `-tls1_2`, and the indications for the obsolete versions `-tls1_1`, `-tls1`, `-ssl3`. It is also possible to forbid certain protocol versions using the flags of the form `-no_[version]`, such as `-no_tls1_1`, `-no_ssl3`, etc. Don't mix those two kinds of flags. For example, this command can be used to check if a server still offers obsolete TLS versions (< TLS 1.2):

```
$ openssl s_client -brief -no_tls1_3 -no_tls1_2 \
  -connect paedubucher.ch:443 -crlf </dev/null
CONNECTION ESTABLISHED
Protocol version: TLSv1.1
Ciphersuite: ECDHE-RSA-AES256-SHA
Peer certificate: CN = paedubucher.ch
```

```
Hash used: MD5-SHA1
Signature type: RSA
Verification: OK
Supported Elliptic Curve Point Formats: uncompressed:ansiX962_compressed_...
Server Temp Key: X25519, 253 bits
DONE
```

In the case above, TLS 1.1 is still offered.

TLS 1.2 ciphers and TLS 1.3 cipher suites can be defined using the `-cipher` and `-ciphersuites`, respectively:

```
$ openssl s_client -brief -cipher TLS_RSA_WITH_AES_128_CBC_SHA256 \
  -connect paedubucher.ch:443 -crlf </dev/null
Error with command: "-cipher TLS_RSA_WITH_AES_128_CBC_SHA256"
140339066332544:error:1410D0B9:SSL routines:SSL_CTX_set_cipher_list:...

$ openssl s_client -brief -cipher ECDHE-RSA-AES256-GCM-SHA384 \
  -connect paedubucher.ch:443 -crlf </dev/null
CONNECTION ESTABLISHED
Protocol version: TLSv1.2
Ciphersuite: ECDHE-RSA-AES256-GCM-SHA384
Peer certificate: CN = paedubucher.ch
Hash used: SHA256
Signature type: RSA-PSS
Verification: OK
Supported Elliptic Curve Point Formats: uncompressed:ansiX962_compressed_...
Server Temp Key: X25519, 253 bits
DONE

$ openssl s_client -brief -tls1_3 -ciphersuites TLS_AES_128_GCM_SHA256 \
  -connect mozilla.org:443 -crlf </dev/null
CONNECTION ESTABLISHED
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_128_GCM_SHA256
Peer certificate: CN = mozilla.org
Hash used: SHA256
Signature type: RSA-PSS
Verification: OK
Server Temp Key: X25519, 253 bits
DONE
```

Use `openssl ciphers` or ciphersuite.info to find proper ciphersuite indications.

# Chapter 3: Certificates

TLS uses X.509 certificates, which is an ITU standard for digital certificates built on ASN.1 (Abstract Syntax Notation One), a cross-platform tree-like data structure with object identifiers (OID).

The X.500 directory standard is used to specify informations about the certificate holder (organization unit `OU=`, organization `O=`, common name `CN=`, etc.) The common name used to be the host name, but can be any identification (uid, email, first and last name).

A *trust anchor* or *root certificate* is an ultimately trusted certificate, often self-signed by some big organization that runs its own Certificate Authority (CA). Those certificates are included in operating systems (usually Mozilla's bundle in Unix-like systems, or Microsoft's bundle in Windows). Those bundles can be curated manually, which causes a lot of work and trouble.

On Unix-like systems, certificates are usually stored under `/etc/ssl/certs`. Use `openssl` to figure out the real and optional additional paths:

```
$ openssl version -a
OpenSSL 1.1.1k  25 Mar 2021
[...]
OPENSSLDIR: "/etc/ssl"
ENGINESDIR: "/usr/lib/engines-1.1"
[...]
```

### Certificate Validation and Verification

All certificates are validated against those in the trust bundles. Additional certificates can be added (and removed) by operating system or distribution specific tools, such as `certctl`, `add-trusted-cert`, `update-ca-trust` etc.

Use the `-CAfile` flag to validate a certificate against a specific CA:

```
$ openssl s_client -verify_return_error -connect www.srf.ch:443 \
  -CAfile /etc/ssl/certs/DigiCert_Global_Root_CA.pem </dev/null >/dev/null
Global_Root_CA.pem </dev/null >/dev/null
depth=2 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert ...
verify return:1
depth=1 C = US, O = DigiCert Inc, CN = DigiCert SHA2 Secure Server CA
verify return:1
depth=0 C = CH, ST = Z\C3\BCrich, L = Z\C3\BCrich,
        O = Schweizer Radio & Fernsehen, CN = *.srf.ch
verify return:1
DONE
```

```
$ echo $?
0
```

A certificate contains two main pieces:

1. information about the entity being certified
2. the digital signature of that information

The signed organization information is put together with a public key into a Certificate Signing Request (CSR). The CSR, which is a certificate without a digital signature, is then submitted to the Certificate Auhtority, which verifies the submitted information more or less thoroughly, and then signs the certificate with its private key for a duration of usually 3 to 12 months. (Modern browsers impose a limit of 398 days.)

Certificates can be constrained:

- Some can be used to sign other certificates within the same domain name.
- The cryptographic algorithms to be used can be constrained.
- The certificate is valid only for a certain domain (`foobar.com`) or, in case of a wildcard certificate, all subdomains thereof (`*.foobar.com`).

Those constraints can be extended beyond the standard. There are critical and non-critical extensions. Critical extensions must be processed and validated by all clients. Non-critical extensions can be processed if the client wants to and is available to.

There are different levels of validation for a certificate:

1. *Domain Validation* (DV): The CA checked that the domain is under control of the requesting entity (usually done via DNS).
2. *Organization Validation* (OV): The CA verified that the requesting organization exists and is located at the address indicated.
3. *Extended Validation* (EV): The CA verifies the business registration. This is expensive; the CA will charge for the certificate accordingly.

Domain Validation is usually enough. Extended Validation is mostly used for regulatory compliance, say, in the finance sector. The requesting entity has to prove its identity in all cases, only the mechanisms differ.

## Chain of Trust

The verification process of a certificate is based on a *Chain of Trust*, which nowadays is rather a *Tree of Trust*. Root CAs protect their private keys very well and don't want to use it for every certificate to be signed. Instead, they sign certificates of intermediate CAs (with lower lifetimes and limited rights), which in turn sign certificates using their private key.

The validation is performed bottom-up: domain owner, intermediate CA, root CA. This requires the whole chain of certificates being available to the client, which only knows the public keys of some well-known root CAs. Therefore, the certififaces and public keys of the intermedia CA must also be delivered in a *CA bundle*.

Certificates can be *cross signed*, i.e. be signed using signatures of different CAs (both intermediate and root). Only one single path from the domain certificate up to the root certificate must be found for a successful validation. This makes a certificate more robust in case an intermediary/root certificate is revoked (see RFC 5280 for details on certificate revocation).

## Certificate Formats

Certificates are usually delivered in the X.509 format, but can be stored in a different formats.

### DER: Distinguished Encoding Rules

*Distinguished Encoding Rules* (DER) is an old binary format using a subset of ASN.1, each information being stored with a tag, a length, and the actual data. This format is very small and usually stored in files with the ending `.der` in their name:

```
$ openssl x509 -in certificate.der -inform der -text -noout
```

### PEM: Privacy-Enhanced Mail

*Privacy-Enhanced Mail* (PEM) is a standard for sending encrypted email, which is nowadays less popular than PGP. It is still in common use to encode keys and certificates. PEM is basically base64-encoded DER with human friendly headers and footers separating multiple certificates or keys:

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

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

Usually, `.pem` is used for the file name ending, byt `.crt` or `.key` is also common, often for backwards compatibility (i.e. when still relying on the old name, even though the transition to a new format has been made). The PEM format is assumed by default, so no `-inform` option needs to be passed in order to read PEM Files:

```
$ openssl x509 -in chain.pem -noout -text
```

Certificates can be re-encoded by combining the the `-in`, `-out`, `-inform`, and `-outform` options. Here, a DER-encoded certificate is converted to the PEM format:

```
$ openssl x509 -in part.pem -inform pem -outform der -out part.der
```

### PKCS#12: Public Key Cryptography Standard 12

*Public Key Cryptography Standard 12* (PKCS#12) can store multiple related encryption files in a single archive, which can be signed and/or encrypted (e.g. a certificate chain combined with a private key). Each piece of information is stored in its own SafeBox, which are combined to archives, usually stored with the ending `.p12` or the older `.pfx`. The `pkcs12` subcommand is used to process such archives. A private key can be combined with a (PEM) certificate as follows:

```
$ openssl pkcs12 -export -out archive.p12 -inkey private.key -in cert.pem
```

Additional certificates can be provided using the `-certfile` option. A password is prompted to encrypt the archive. A PKCS#12 file can be viewed as follows:

```
$ openssl pkcs12 -info -in archive.p12
```

Pass the `-nodes` option to omit encryption for the private key in cleartext, `-nokeys` to omit any keys in the output, and `-nocerts` to omit the certificates. Those options can be combined to split up a PKCS#12 archive into certificate and private key files:

```
$ openssl pkcs12 -in archive.p12 -out all.crt -nodes
$ openssl pkcs12 -in archive.p12 -out certs.crt -nokeys
$ openssl pkcs12 -in archive.p12 -out private.key -nocerts -nodes
```

Notice that the output file `private.key` in the last example is exported in the PKCS#8 format, i.e. without an algorithm mentioned in the header:

```
-----BEGIN PRIVATE KEY-----
...
-----END PRIVATE KEY-----
```

Pipe the output through `openssl rsa` or `openssl ec` in order to transform it to the PKCS#1 format with an algorithm indication:

```
$ openssl pkcs12 -in archive.p12 -nocerts -nodes | openssl rsa -out key.pem
```

Notice that the common endings `.pem`, `.der`, and `.crt` do not necessarily imply the format used; better rely on the output of `file(1)` and the validation of `openssl-x509(1ssl)` instead.

### Certificate Contents

A certificate contains various fields, which can be viewed as follows (output of public keys and signatures shortened):

```
$ openssl x509 -in cert.der -inform der -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            04:a9:5c:4e:9c:51:cd:df:b3:ef:00:78:5b:97:b5:7f:79:39
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Let's Encrypt, CN = R3
        Validity
            Not Before: Apr 26 08:25:39 2021 GMT
            Not After : Jul 25 08:25:39 2021 GMT
        Subject: CN = paedubucher.ch
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:a0:98:97:a0:9d:41:4d:3a:27:2d:c3:86:12:ce:
                    ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
             TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
               44:9F:81:5F:58:39:34:C1:0C:E1:A0:E1:3E:B0:BF:E2:61:12:...
            X509v3 Authority Key Identifier:
               keyid:14:2E:B3:17:B7:58:56:CB:AE:50:09:40:E6:1F:AF:9D:...
```

```
        Authority Information Access:
            OCSP - URI:http://r3.o.lencr.org
            CA Issuers - URI:http://r3.i.lencr.org/

        X509v3 Subject Alternative Name:
            DNS:paedubucher.ch, DNS:www.paedubucher.ch
        X509v3 Certificate Policies:
            Policy: 2.23.140.1.2.1
            Policy: 1.3.6.1.4.1.44947.1.1.1
              CPS: http://cps.letsencrypt.org

        CT Precertificate SCTs:
            Signed Certificate Timestamp:
                Version   : v1 (0x0)
              Log ID    : 94:20:BC:1E:8E:D5:8D:6C:88:73:1F:82:8B:...
                          D1:DA:4D:5E:6C:4F:94:3D:61:DB:4E:2F:58:...
                Timestamp : Apr 26 09:25:39.415 2021 GMT
                Extensions: none
                Signature : ecdsa-with-SHA256
                          30:44:02:20:0B:2F:D4:47:A0:86:F4:9E:F0:...
                            ...
            Signed Certificate Timestamp:
                Version   : v1 (0x0)
              Log ID    : F6:5C:94:2F:D1:77:30:22:14:54:18:08:30:...
                          E3:4D:13:19:33:BF:DF:0C:2F:20:0B:CC:4E:...
                Timestamp : Apr 26 09:25:39.392 2021 GMT
                Extensions: none
                Signature : ecdsa-with-SHA256
                          30:45:02:20:4D:7C:04:F4:F7:02:BC:3F:2B:...
                            ...
    Signature Algorithm: sha256WithRSAEncryption
         60:1a:51:cc:77:4c:5d:f7:31:9a:f3:93:31:5c:74:19:3e:70:
         ...
```

- `Version` (usually 3) is the X.509, *not* the TLS version.
- `Serial Number` is a unique number, which is useful for certificate revocation.
- `Signature Algorithm` describes how the CA signed the certificate.
- `Issuer` identifies the CA that issued the certificate.
- `Validity` defines a time span in which a certificate can be used.
- `Subject` contains information about the entity being certified.
  - For Domain Validation (DV), only the common name (CN) is listed.
  - For Organization or Extended Validation (OV and EV), information about the organization, city, country are listed.

- `Subject Public Key` is the public part of the key that has been used to create the Certificate Signing Request (CSR).
- `X509v3 extensions` lists critical and non-critical extensions (mandatory and optional for certificate verification):
  - `X509v3 Key Usage` (critical) describes how a key can be used.
  - `X509v3 Extended Key Usage` (non-critical) describes additional purposes the key can be used for.
  - `X509v3 Basic Constraints` (critical) lists if the certificate's key can be used to sign other certificates (`CA:TRUE`) or not (`CA:FALSE`)
  - `X509v3 Subject Key Identifier` and `X509v3 Authority Key Identifier` is the identifier for the subject's and the CA authority's key.
  - `Authority Information Access` shows how to get more information about the CA.
  - `X509v3 Subject Alternative Name` shows the hostnames covered by the certificate.
  - `X509v3 Certificate Policies` describes the CA.
- `CT Precertificate SCTs` contains the Signed Certificate Timestamp (SCT) with the signatures used, which is a cryptographic proof that the certificate was submitted to a certificate log (Certificat Transparency).
- There's a digital signature of the CA at the very end of the certificate with the indicated `Signature Algorithm`.

### Narrowing Down the Output

Additional information to X.509 extensions can be queried using the `-ext` option with comma-separated extensions to be listed (`x509v3_config(3)`):

```
$ openssl x509 -in cert.pem -noout -ext keyUsage,extendedKeyUsage
X509v3 Key Usage: critical
    Digital Signature, Key Encipherment
X509v3 Extended Key Usage:
    TLS Web Server Authentication, TLS Web Client Authentication
```

Extensions not understood by the `openssl` client in use are displayed as raw binary data (consider updating `openssl` or look up the respective OID).

The output can be further shortened using the `-certopt` option, which accepts comma-separated values (here: neither display public keys nor signature dumps):

```
$ openssl x509 -in first.pem -text -noout -certopt no_pubkey,no_sigdump
Certificate:
```

```
Data:
    Version: 3 (0x2)
    Serial Number:
        04:a9:5c:4e:9c:51:cd:df:b3:ef:00:78:5b:97:b5:7f:79:39
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C = US, O = Let's Encrypt, CN = R3
    Validity
        Not Before: Apr 26 08:25:39 2021 GMT
        Not After : Jul 25 08:25:39 2021 GMT
    Subject: CN = paedubucher.ch
    X509v3 extensions:
        X509v3 Key Usage: critical
            Digital Signature, Key Encipherment
        X509v3 Extended Key Usage:
            TLS Web Server Authentication, TLS Web Client Authentication
        X509v3 Basic Constraints: critical
            CA:FALSE
        X509v3 Subject Key Identifier:
            44:9F:81:5F:58:39:34:C1:0C:E1:A0:E1:3E:B0:BF:E2:61:12:...
        X509v3 Authority Key Identifier:
            keyid:14:2E:B3:17:B7:58:56:CB:AE:50:09:40:E6:1F:AF:9D:...

        Authority Information Access:
            OCSP - URI:http://r3.o.lencr.org
            CA Issuers - URI:http://r3.i.lencr.org/

        X509v3 Subject Alternative Name:
            DNS:paedubucher.ch, DNS:www.paedubucher.ch
        X509v3 Certificate Policies:
            Policy: 2.23.140.1.2.1
            Policy: 1.3.6.1.4.1.44947.1.1.1
              CPS: http://cps.letsencrypt.org

        CT Precertificate SCTs:
            Signed Certificate Timestamp:
                Version   : v1 (0x0)
                Log ID    : 94:20:BC:1E:8E:D5:8D:6C:88:73:1F:82:8B:...
                            D1:DA:4D:5E:6C:4F:94:3D:61:DB:4E:2F:58:...
                Timestamp : Apr 26 09:25:39.415 2021 GMT
                Extensions: none
                Signature : ecdsa-with-SHA256
                            30:44:02:20:0B:2F:D4:47:A0:86:F4:9E:F0:...
                            ...
            Signed Certificate Timestamp:
```

```
Version   : v1 (0x0)
Log ID    : F6:5C:94:2F:D1:77:30:22:14:54:18:08:30:...
            E3:4D:13:19:33:BF:DF:0C:2F:20:0B:CC:4E:...
Timestamp : Apr 26 09:25:39.392 2021 GMT
Extensions: none
Signature : ecdsa-with-SHA256
            30:45:02:20:4D:7C:04:F4:F7:02:BC:3F:2B:...
            ...
```

### Multiple Hostnames

A site can be reachable under different names, such as `paedubucher.ch` and `www.paedubucher.ch`. Subject Alternative Names (SAN) identify all the hostnames a certificate is good for and can be displayed as follows:

```
$ openssl x509 -in first.pem -noout -ext subjectAltName
X509v3 Subject Alternative Name:
    DNS:paedubucher.ch, DNS:www.paedubucher.ch
```

A wildcard certificate is good for any subdomain of a hostname (`*.paedubucher.ch`), which can be dangerous, because a successful attacker can offer services with subdomains made up for the purpose (such as `www2.paedubucher.ch`).

### Fetching Certificates

It's also possible to fetch and display remote certificates using the `s_client` and `x509` sub-commands combined with a pipe (same output as further above):

```
$ openssl s_client -connect paedubucher.ch:443 </dev/null | \
  openssl x509 -text -noout -certopt no_pubkey,no_sigdump
```

Use the `-showcerts` option to display the whole certificate chain:

```
$ openssl s_client -showcerts -connect paedubucher.ch:443 </dev/null
```

### Some CA Considerations

When buying a certificate, consider the reputation a CA has, and in which jurisdication it is located. A CA must support Certificate Revocation Lists (CRL), the Online Certificat Status Protocol (OCSP), and, optionally, Certification Authority Authorization (CAA).

# Chapter 4: Revocation and Invalidation

When private key gets stolen, the certificates based on it can no longer be trusted. However, the certificate itself still looks trustworthy, and cannot be revoked by the owner of the private key that has been used to sign the certificate. Since trust comes from above, only the signing CA can revoke a certificate.

CAs offer web interfaces or automated tools and interfaces for this purpose. Usually, a replacement certificate is ordered in this purpose, which of course needs to be requested with a new private key. It's a good idea to test this process, which gives you an idea how well the CA can handle the certificate invalidation and replacement process.

TLS offers multiple mechanisms for certificate revocation:

## Certificate Revocation Lists (CRL)

*Certificate Revocation Lists* (CRL) are lists of revoked (but not yet expired) certificates offered by the CA. An endpoint to this list is linked as the `CRL Endpoint` (`Distribution Point`) in the CA's intermediary certificate. The client downloads this list and makes sure the respective certificate is not on that list by comparing the certificate's serial number to those on the list.

CRLs become big quite fast and don't scale very well nowadays, even though they can be cached. Caching, however, slows down the revocation process. Use the `crl` subcommand on your CA's root certificate to show the CRL:

```
$ curl http://crl.identrust.com/DSTROOTCAX3CRL.crl | \
  openssl crl -text -inform der -noout
```

CRLs are usually served in DER format to keep the files small, but other formats can be used, too.

## Online Certificate Status Protocol (OCSP)

With the *Online Certificate Status Protocol* (OCSP), the client no longer needs to fetch a CA's complete list of rekoved certificates, but can query the status of a single certificate via an HTTP endpoint. A result (`good`, `revoked`, `unknown`) and a cache time to live are returned. The CA signs the response, so raw HTTP (without TLS) is used here. The endpoint for OCSP can be extracted from the certificate chain:

```
$ openssl s_client -showcerts -connect paedubucher.ch:443 \
  </dev/null 2>/dev/null | openssl x509 -noout -ocsp_uri
http://r3.o.lencr.org
```

Given the certificate chain and the OCSP URL, the revocation status can be tested using the `ocsp` subcommand (`openssl-ocsp(1ssl)`):

```
$ openssl s_client -connect github.com:443 </dev/null | \
  openssl x509 >cert.pem
$ openssl s_client -showcerts -connect github.com:443 \
  </dev/null >chain.pem
$ openssl x509 -in chain.pem -noout -ocsp_uri
http://ocsp.digicert.com
$ openssl ocsp -issuer chain.pem -cert cert.pem -text \
  -url http://ocsp.digicert.com
OCSP Response Data:
OCSP Response Status: successful (0x0)
Response Type: Basic OCSP Response
Version: 1 (0x0)
Responder Id: 5061A6A0D235C4112A208D1F0FAC42F0CD29CF4B
Produced At: Jun 24 03:36:53 2021 GMT
Responses:
Certificate ID:
  Hash Algorithm: sha1
  Issuer Name Hash: C6325AEE2FA3FD33D07789FD6B4CCEF0CA3FD029
  Issuer Key Hash: 5061A6A0D235C4112A208D1F0FAC42F0CD29CF4B
  Serial Number: 0E8BF3770D92D196F0BB61F93C4166BE
Cert Status: good
This Update: Jun 24 03:21:02 2021 GMT
Next Update: Jul  1 02:36:02 2021 GMT
```

Check the `Cert Status`; "good" means that the certificate hasn't been revoked.

This process consumes less bandwith than CRLs, but more processing power on the client side. The CA also gets to know the clients accessing particular domains, which is a privay issue.

### OCSP Stapling

The OCSP query response (see above) contains a field `Next Update`, which is the expiration date of that query. A server can make this request on behalf of the client, and attach ("staple") the OCSP response to the TLS session with the client, and digitally sign it. Doing so, the server can save the clients a lot of OCSP queries—and better protect their privacy—but also needs to perform the OCSP lookups periodically (according to the `Next Update` indication). Most modern web browsers and servers support OCSP nowadays.

### Revocation Issues

Not all CAs offer all the revocation mechanisms described, or they implement them in a non-standard or bad way (say, offering just empty CRLs for technical compatibility). Client software failing to perform OCSP checks are often hidden from the user for the sake of convenience. Some modern browsers rely on their own curated list of CRLs, which are shipped with their software updates, instead of fetching CRLs in real-time.

How a client deals with revocation can be tested with sites like revoked-rsa-dv.ssl.com. Using short-lived certificates with heavy automation mitigates revocation issues. Unfortunately, Chrome ignores the OCSP Must Staple server setting, which would be a way to emulate short-lived certificates.

## Chapter 5: TLS Negotiation

Clients and servers may have different software and configurations for TLS deployed, which support different protocol versions, algorithms, and options. Therefore, the parameters to be uses for a connection are not known in advance, but need to be negotiated between client and server.

A TLS connection, which can be initiated using the `s_client` subcommand, consists of three parts: certificate validation, protocol settings, and session resumption.

### Certificate Validation

The TLS client attempts to find a way from the served certificate up to a trusted root certificate. The process is finished as soon as one such valid path is discovered. The `openssl` client then then outputs this path to standard error:

```
$ openssl s_client -connect paedubucher.ch:443 </dev/null >/dev/null
depth=2 C = US, O = Internet Security Research Group, CN = ISRG Root X1
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = R3
verify return:1
depth=0 CN = paedubucher.ch
verify return:1
DONE
```

The certificates are listed from top (root certificate) to bottom (domain certificate), with a `depth` field indicating the distance from the domain certificate. The field `verify return:1` signifies successful validation of the certificate.

The certificate chain is displayed in reverse order (from domain to root) in the standard output, followed by the server certificate, details about the algorithms and keys being used (here: `SHA256`, `RSA-PSS`, and `X25519` with 253 bits), and, finally, the result of the SSL handshake (`Verification: OK`):

```
$ openssl s_client -connect paedubucher.ch:443 </dev/null 2>/dev/null
...
Certificate chain
 0 s:CN = paedubucher.ch
   i:C = US, O = Let's Encrypt, CN = R3
 1 s:C = US, O = Let's Encrypt, CN = R3
   i:C = US, O = Internet Security Research Group, CN = ISRG Root X1
 2 s:C = US, O = Internet Security Research Group, CN = ISRG Root X1
   i:O = Digital Signature Trust Co., CN = DST Root CA X3
...
-----BEGIN CERTIFICATE-----
MIIFNjCCBB6gAwIBAgISBJa0Pa3QlqD/TEZHmLlZtqQnMA0GCSqGSIb3DQEBCwUA
...
wnvurz8wdWtXilw61qAJJwivHeU+/FfF1Lt+wRN6mDZ/bQoU3dFubw1n
-----END CERTIFICATE-----
subject=CN = paedubucher.ch

issuer=C = US, O = Let's Encrypt, CN = R3
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 4691 bytes and written 409 bytes
Verification: OK
---
```

### Protocol Settings

Parameters such as key length, TLS version and cipher, and the like are negotiated between the parties involved. Compression, being deactivated by default, can be activated using the `-comp` flag.

Application Layer Protocol Negotiation (ALPN) is a way to integrate TLS setup into the protocol setup, which is mostly used in HTTP/2, and can be activated using the `-alpn` flag.

Application data can be bundled with a TLS connection using the `-early_data` flag. All those informations are displayed in the subsequent sections of the output:

```
$ openssl s_client -connect paedubucher.ch:443 </dev/null 2>/dev/null
...
New, TLSv1.2, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
```

### Session Resumption

The TLS session and resumption details vary strongly between TLS versions being used:

#### TLS 1.2

The session information starts with the protocol version (`TLSv1.2`) and the cipher being used. Every session has its ID (`Session-ID`) and Context ID (`Session-ID-ctx`), which could refer to some server-internal context such as an application (database server, web server) and is often used for load balancing.

The `Master Key` is the result of the key agreement between client and server. Additional fields (prefixes: `PSK` and `SRP`) are only set if pre-shared keys and the Secure Remote Password (SRP) protocol are being used.

The actual *session ticket* follows, which is then used for a subsequent request within the same TLS session (*resumption*).

The ticket is accompanied by TTL information indicating the timespan in which the ticket is valid. `Verify return code: 0 (ok)` signifies success, all other codes point to a verification error.

```
$ openssl s_client -tls1_2 -connect paedubucher.ch:443 </dev/null 2>/dev/null
...
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES256-GCM-SHA384
    Session-ID: 3E09DF7A7C7C90B577C12254ED7ACA74CCCB5BDBE5B5B2AF44B...
    Session-ID-ctx:
    Master-Key: BB1EA0160136226FEB81B5849161EA9C87BAE06EAFAF649722A...
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 300 (seconds)
```

```
TLS session ticket:
0000 - 5d 43 c4 64 d9 18 b1 bd-c7 42 cc e9 49 44 2a fd   ]C.d...
...
00b0 - 77 9f 38 b6 a2 cf bd 03-e6 7f 31 e1 f7 5f 58 b0   w.8...

Start Time: 1624727116
Timeout   : 7200 (sec)
Verify return code: 0 (ok)
Extended master secret: yes
```

#### TLS 1.3

TLS 1.3 supports many fields of TLS 1.2 just for the sake of backward compatibility, which is important for network devices and tools that perform deep packet inspection on network traffic.

In TLS 1.3, sessions are only established *after* the main handshake has been completed. Therefore, no `SSL-Session` section can be found when dealing with TLS 1.3.

### TLS Failures

A TLS connection usually fails for two reasons:

1. The client won't accept a certificate.
2. Client and server cannot agree on TLS options, algorithms, and protocols.

If the client uses a current software version and certificate bundle together with a default configuration, then usually the server is to blame for a failed TLS connection.

One common error is a server only serving its own domain certificate instead of providing the whole certificate chain. This usually results in an error message like "unable to get local issuer certificate". Searching for an OpenSSL error message usually yields quick and accurate results.

The website badssl.com provides many subdomains with different TLS and SSL (mis)configurations. Try them out with the `s_client` subcommand, and make sure to pass the `-verify_return_error`, so that you'll learn which TLS issues cause which OpenSSL error messages.

```
$ openssl s_client -connect untrusted-root.badssl.com:443 \
  -verify_return_error </dev/null 2>/dev/null | grep 'Verification error'
Verification error: self signed certificate in certificate chain

$ openssl s_client -connect superfish.badssl.com:443 \
  -verify_return_error </dev/null 2>/dev/null | grep 'Verification error'
Verification error: unable to get local issuer certificate
```

# Chapter 6: Certificate Signing Requests and Commecrial CAs

Certificate Signing Requests (CSR) are specified in RFC 2986. They contain all the information that is verified by the CA, and then signed. A CSR can be seen as an unsigned certificate. It is a good idea to automate the process of creating CSRs, so that you get it right without re-trying multiple times, especially if you urgently need to replace a certificate based on a private key that just has been leaked. It's a good idea to create a new private key with each request, because older private keys are more likely to have been leaked unknowingly, and they also tend to base on older best practices and cryptographic algorithms.

## Gathering Information

Free CAs (like ACME) are a good choice for DV (domain validation) certificates. Internal policy, the need for a OV (organization validation) or EV (extended validation), or if you want to run your own CA are reasons to use a commercial CA instead. DV certificates usually only require a domain name. More information is needed for an OV or EV certificate. Make sure to gather this information in advance:

- Country Name (`C`): two-letter country code (ISO 3166), e.g. `CH` for Switzerland
- State/Provice (`ST`): spelled out name of state or provice, e.g. `Lucerne`
- Locality (`L`): the city name, in which a company is *officially* located, e.g. `Lucerne`
- Organization (`O`): the company's legal name, e.g. `Foo Brewery AG` (not just `Foo Brewery`!)
- Organization Unit (`OU`): the department that handles the certificates (usually `IT`), optional field

Storing the hostname under Common Name (`CN`) is an obsolete practice, which is still used a lot for the sake of backward compatibility. Notice that the `CN` field is limited to 63 characters, and no name constraints are checked for this field by the CA.

## RSA or ECDSA

There are two possible choices for a public key algorithm:

1. RSA is the time-proofen option that is supported by most CAs and most software.
2. ECDSA is a newer standard that provides the same security with shorter key lengths.

Consider ECDSA to save processing power for cryptography if the certificates are mostly used on devices with little computing power or that run on battery. It is possible that the CA's root certificate uses a different algorithm than the CSR, but then the client has to deal with both RSA and ECDSA. Better pick a CA that supports your choice of algorithm. It is also possible to deploy one certificate by algorithm, depending on the software you're using.

### OpenSSL Configuration

The information needed for a CSR can be entered in different ways:

1. Using an interactive prompt.
2. Using command-line flags.
3. Using configuration files.

Using the interactive prompt is very error-prone and limits automation. Better use one of the other two options.

In order to provide your CSR details with a configuration file, you can get to know the default configuration of your local OpenSSL installation:

```
$ openssl version -a | grep -i openssldir
OPENSSLDIR: "/etc/ssl"
```

The configuration is located in that directory under `openssl.cnf`, so in this example in `/etc/ssl/openssl.cnf`. The file is organized in different section. For example, configuration relevant for the `req` subcommand is stored under the `[ req ]` section. The settings are stored as key-value pairs. Comments start with `#` and go to the end of a line. Better do *not* modify this file, but provide local files for specific needs.

### CSR Configuration File

When creating a CSR, make sure to name the files used for this purpose properly, i.e. containing the domain name, for example `paedubucher.ch-private.key` for a private key, `paedubucher.ch.csr` for the CSR, and `paedubucher.ch.crt` for the certificate you get back from the CA.

The CSR is created using openssl's `req` subcommand. Let's gather the required information in a config file (`paedubucher.ch.conf`):

```
[ req ]
prompt             = no
default_keyfile    = paedubucher.ch-private.key
distinguished_name = req_distinguished_name
req_extensions     = v3_req
encrypt_key        = yes
output_password    = topsecret

[ req_distinguished_name ]
C  = CH
```

```
ST = Lucerne
L  = Lucerne
O  = Patrick Bucher Kompooter AG
OU = Department of IT Operations
CN = paedubucher.ch

[ v3_req ]
subjectAltName = DNS:paedubucher.ch,DNS:www.paedubucher.ch
```

There are three sections with the following options:

- `req`: parameters to create the CSR
  - `prompt`: whether (`yes`) or not (`no`) to prompt information interactively from the command line
  - `default_keyfile`: path to the file the new private key is stored in
  - `distinguished_name`: pointing to the section the information to be validated is stored
  - `req_extensions`: pointing to extensions used (here: SAN stored in X.509v3 extension)
  - `encrypt_key`: whether (`yes`) or not (`no`) the private key should be encrypted with a password (the command line option `-nodes` for "no DES" deactivates encryption, even though DES is no longer used for that purpose)
  - `output_password`: the password in plain text to encrypt the private key with, omit if `encrypt_key = no`
- `req_distinguished_name`: a section containing the information to be validated by the CA (see the meaning of those fields further above)
- `v3_req`: information for X.509v3 extensions.
  - `subjectAltName`: list all the (sub)domains for which this certificate should be valid as comma-separated values with `DNS:` prefix (see RFC 5280)

When using an encrypted private key, the password needs to be provided when the service using the certificate is restarted. Protect the config file containing the password as good as the private key!

If a lot of subject alt names are to be defined (say, more than would fit on a single line), you can use an array instead:

```
[ v3_req ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = paedubucher.ch
```

```
DNS.2 = www.paedubucher.ch
DNS.3 = *.cdn.paedubucher.ch
```

The entries have to be listed with increasing numbers (`i` in `DNS.i`), gaps are allowed.

## Creating the CSR

The CSR, which should be stored in a file named `[domain].csr`, can be created using openssl's `req` subcommand. Notice that there are some differences depending on the public key algorithm to be used.

### ECDSA

When using ECDSA, a parameters file to configure the elliptic curve is needed.

First, pick one among the available curves, which can be listed using the `ecparam` subcommand:

```
$ openssl ecparam -list_curves
```

`P-256` (`prime256v1`) is a good default choice.

Second, create the parameters file to configure the curve using the `genpkey` subcommand:

```
$ openssl genpkey -genparam -out ec-p256-params.pem -algorithm ec \
  -pkeyopt ec_paramgen_curve:prime256v1
```

Finally, the CSR can be created:

```
$ openssl req -newkey ec:ec-p256-params.pem -config paedubucher.ch.conf \
  -out paedubucher.csr
```

Which should create two files: the private key `paedubucher.ch-private.key` and the actual CSR `paedubucher.csr`.

### RSA

When using RSA, the configuration file needs to be modified by including the `default_bits` (usually `2048` or `4096`) and the hashing algorithm (e.g. `sha256`):

```
[ req ]
prompt             = no
default_bits       = 2048
default_md         = sha256
# same as above...
```

Since the crypto parameters are already provided in the configuration file, the CSR can be created without further ado:

```
$ openssl req -newkey rsa -config paedubucher.ch.conf -out paedubucher.csr
```

Again, the private key `paedubucher.ch-private.key` and the actual CSR `paedubucher.csr` should have been generated.

### Client Certificates

The information needed for client certificates mostly depends on the application requesting such a certificate. Here's a config file (`application.conf`) for a client certificate supposed to verify a subject by email using RSA encryption:

```
[ req ]
prompt             = no
default_bits       = 2048
default_md         = sha256
default_keyfile    = application-private.key
distinguished_name = req_distinguished_name
encrypt_key        = yes
output_password    = topsecret

[ req_distinguished_name ]
CN = Patrick Bucher
emailAddress = patrick.bucher@mailbox.org

[ v3_req ]
subjectAltName = email:patrick.bucher@mailbox.org
```

The CSR is created as follows:

```
$ openssl req -newkey rsa -config application.conf -out application.csr
```

Which creates two files: `application-private.key` and `application.csr`.

For applications like VPN connections, a passphrase is commonly used, since the key lies on a client device. Other applications do without a passphrase.

Notice that when leaving a way a distinguished name (`DN`) for an OV or EV certificate, the `prompt` setting cannot be set to `no` via configuration file. Use the `-subj` flag with the value `/` to suppress the prompt nonetheless.

### Without Config File

If you need to create your CSR without an intermediary config file, you can provide all the information needed using the `-subj` and `-addext` command line flags:

```
$ openssl req -newkey rsa:2048 \
  -keyout paedubucher.ch-private.key \
  -out paedubucher.ch.csr \
 -subj '/C=CH/ST=Lucerne/L=Lucerne/O=My Company/OU=IT/CN=paedubucher.ch' \
  -addext 'subjectAltName=DNS:paedubucher.ch,DNS:www.paedubucher.ch'
```

Use `-nodes` for an unprotected private key file. Make sure to provide the `C`, `ST`, `L`, and `O` field for OV and EV certificates. DV certificates only require the `CN` field.

### Viewing a CSR

Before sending the CSR to your CA, double check it:

```
$ openssl req -in paedubucher.ch.csr -noout -text
Certificate Request:
    Data:
        Version: 1 (0x0)
       Subject: C = CH, ST = Lucerne, L = Lucerne, O = My Company, OU = ...
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:c4:c8:a3:c7:c4:87:67:8b:80:04:b7:c7:b9:01:
                    ...
```

```
            Exponent: 65537 (0x10001)
        Attributes:
        Requested Extensions:
            X509v3 Subject Alternative Name:
                DNS:paedubucher.ch, DNS:www.paedubucher.ch
    Signature Algorithm: sha256WithRSAEncryption
         81:32:09:8a:c3:2e:9e:da:a9:5f:7f:f1:60:c2:97:18:1d:92:
         ...
```

Are all the parameters correct? Is there anything misspelled? Store as much information in configuration files and/or scripts as possible, so that re-creating the CSR is only a matter of seconds.

If everything is fine, submit your CSR to your CA.

## Storing the Certificate

When you get your certificate back from the CA, make sure to store it properly on your server:

- Make sure to store the private key with the certificate. A sub-folder for private keys (e.g. `/etc/certs/keys`) is often used with certificates being stored one level above (e.g. `/etc/certs`).
- Those folders should be readable only by `root`. The files therein should be owned by `root` and by a group to which application users belong (e.g. `nginx`).
- Prefix the files stored with the (sub)domain name. Consider adding date information as a prefix, too (e.g. `2021-07-03-paedubucher.ch-private.key`).
- Keep backups of those files.
- Consider storing passphrases in a password manager, which is backed up, too.

### Matching CSR, Private Key, and Certificate

If you don't know which files (CSR, private key, and certificate) belong together, you can figure this out using the *modulus*, which is a cryptographic information that is commonly stored in all the files mentioned.

Use the respective subcommand per file (`x509` for the certificate, `rsa` or `ecdsa` for the private key, and `req` for the CSR) to figure out the modulus using the `-modulus` flag. Pipe the output through the `md5` subcommand for easier comparison:

```
$ openssl x509 -noout -modulus -in paedubucher.ch.crt | openssl md5
(stdin)= 00ae93c0ba0b571cfbe5e6e0233a6fb1
$ openssl rsa -noout -modulus -in paedubucher.ch-private.key | openssl md5
```

```
(stdin)= 00ae93c0ba0b571cfbe5e6e0233a6fb1
$ openssl req -noout -modulus -in paedubucher.ch.csr | openssl md5
(stdin)= 00ae93c0ba0b571cfbe5e6e0233a6fb1
```

Here, certificate, private key, and CSR belong to one another!

It's a good idea to write a script (accepting the domain name `paedubucher.ch` as a parameter) to automate the process.

# Chapter 7: Automated Certificate Management Environment

The *Automated Certificate Management Environment* (ACME) is a protocol for clients interacting with CAs, defined in RFC 8555 by the Internet Security Research Group (ISRG), which run's its own CA called *Let's Encrypt*. The API of ACME can be used for the entire process: creating an account, negotiating the challenges, submitting the CSRs, and deploying the certificates.

## ACME Registration

The ACME client creates a key pair to identify the client. The client contact's the CA's server, accepts its terms and conditions. The server then registers an account identified by the client's public key. This key is then used to sign further interaction between client and server.

Commercial CAs can make use of this key as well, which is linked to a user account. Additional services (EV, OV) are provided against payment by those CAs.

## ACME Challenge Process

Once registered, the client can request certificates for the domains (or hosts) under his control. The server responds with a list of challenge methods (one per domain) the client can use to prove his ownership of the respective domain. The client picks one challenge per domain and reports his choice to the server. Depending on the of challenge, the server provides additional information the client needs to pass the verification. If the challenge succeeded, the client can submit CSRs to the server, which will respond with signed certificates ready for deployment.

### ACME Challenges

An ACME challenge requires the client to place some *key authorization* under a specific *token*. The token is a specific location, and the key authorization combines the token with a digest of the client account's key. If the server can find the specified key authorization by accessing the given token, the client has proven his ownership of the domain successfully.

ACME supports the following challenge methods:

1. **HTTP-01**: The server verifies if the client controls a web server handling the domain certificates are about to be requested for. For the domain `paedubucher.ch`, the key authorization must be made available under `http://paedubucher.ch/.well-known/acme-challenge/[token]`, with the token indicated by the server. The server verifies if the token with the correct content (key authorization) is served from this location (URL). This challenge runs under port 80, but can be redirected to port 443. HTTP-01 is the simplest challenge, but cannot be used for wildcard certificates.
2. **DNS-01**: The server verifies if the client controls the DNS server managing the domain certificates are about to be requested for. For the domain `paedubucher.ch`, a `TXT` record under the token `_acme_challenge.paedubucher.ch` must be provided containing the key authorization as its value. DNS-01 is especially useful for web servers not facing the Internet, and if wildcard certificates are being requested. It also makes it possible to request certificates from an other server than they are used on.
3. **TLS-ALPN-01**: Much like HTTP-01, the server verifies that the client controls a server facing the Internet. Unlike HTTP-01, the server need not be a web server, but a TLS-aware server running on port 443 supporting *Application Layer Protocol Negotiation* (ALPN), which allows to run different services under a single port. For this challenge, the client doesn't need any token/key authorization information from the server, but can setup everything before picking the challenge method. It's possible to take the productive web server down and start up a special ALPN server running on the same port during the verification process, which causes some downtime. Web servers support ALPN using modules (Apache's `mod_md`) or by a special proxy configuration (nginx). TLS-ALPN-01 is the right choice for deployments using proxies and/or load balancers, but requires the server(s) to be publicly reachable by the Internet. Wildcard certificates are not supported by this challenge method.

Use DNS-01 if you need a wildcard certificate or a certificate for a server not facing the Internet. Use HTTP-01 if it works for your environment, and pick TLS-ALPN-01 otherwise.

### Some Practical Advice

When first testing and deploying ACME, make sure to not hit your CA's imposed by-account resource limit. If your CA offers a testing or staging environment (which don't provide trusted certificates), try to get your setup right first by using one of these. If everything works, use the productive environment.

After the initial successful deployment, it's a good idea to test the renewal process after two thirds into the certificate's lifetime, i.e. after 60 days for a certificate valid for 90 days. (Some CAs do not renew certificates that are younger. Thirty days is plenty of time left to fix your environment if something doesn't work.)

### ACME Clients

There are plenty of ACME clients to choose from, some of them are:

- OpenBSD's `acme-client(1)`, which, unfortunately, isn't available neatly packed for other operating systems.
- Apache's `mod_md`, which manages ACME right from the web server.
- Docker's *Let's Encrypt* container, which does everything for you.
- The EFF's `certbot`, which was the first ACME implementation, comes with heavy Python dependencies, but doesn't support the TLS-ALPN-01 challenge yet.
- Dehydrated, which is a simple client based on shell scripts and basic system utilities, and therefore should work on any Unix-like environment.

### Dehydrated

Dehydrated can be downloaded and installed using a package manager or manually from dehydrated.io. Make sure `/etc/dehydrated` exists, into which folder the example configuration shipped with dehydrated should be copied (`/etc/dehydrated/config`).

The main script `dehydrated` relies on hook scripts that provide the functionality specific to each challenge. For HTTP-01, `hook.sh` is provided as an example. The Dehydrated web site provides additional hook scripts for other challenges and specific software packages (e.g. DNS servers, load balancers, etc.). Put `hook.sh` into your configuration path (`/etc/dehydrated/hook.sh`). (Alternatively, modify your configuration so that it points to this hook script.)

Certificates created by Dehydrated should only be accessible by `root`—and the user that runs the dehydrated scripts. Create an unprivileged user called `acme` with a home in `/var/acme`. This is where certificates are going to be stored. Do not allow the user to login by setting a bogus shell (e.g. `/usr/bin/nologin`). Also set a lengthy password. (Remember: this setup takes place on a Internet-facing server, which is prone to attacks from the outside.) On Arch Linux:

```
# useradd -d /var/acme -m -s /usr/bin/nologin -U acme
# chown -R acme:acme /var/acme
# passwd acme
```

Modify the configuration (`/etc/dehydrated/config`) so that `BASEDIR` points to `/var/acme`—the home directory just set for the `acme` user. Also set the `DEHYDRATED_USER` and `DEHYDRATED_GROUP` to `acme`. Provide a proper `CONTACT_EMAIL` address.

List the domains to manage certificates for in `/etc/dehydrated/domains.txt` (more of which later) and set `DOMAINS_TXT` accordingly.

For a challenge other than HTTP-01, set the challenge type using `CHALLENGETYPE`. Set the `CA` to the certificate authority to be used: `letsencrypt` (default), `letsencrypt-test` (for testing your setup), `buypass`, `buypass-test`, `zerossl` (others supported by default). For other CAs, set `CA` to the API URL provided by the respective CA instead of its name.

Additional configuration settings can be put in an extra folder, say `/etc/dehydrated/config.d` to be referred to by the option `CONFIG_D`.

```
BASEDIR="/var/acme"
DEHYDRATED_USER="acme"
DEHYDRATED_GROUP="acme"
CONTACT_EMAIL="patrick.bucher@mailbox.org"
DOMAINS_TXT="/etc/dehydrated/domains.txt"
CHALLENGETYPE="http-01"
CA="letsencrypt-test"
CONFIG_D="/etc/dehydrated/config.d"
```

The files in `CONFIG_D` ending in `.sh` will be processed in alphanumerical order, with later files overriding settings of earlier files.

Put all the domains that will use the same certificate on a single line in your domain list (`/etc/dehydrated/domains.txt`). Make sure to put the domain with the Common Name (CN) first (max. 64 characters):

```
foo.bar www.foo.bar mail.foo.bar
qux.com www.qux.com mail.qux.com
```

The common name (above: `foo.bar` and `qux.com`) will be used as a directory name to store the certificates inside. Define an optional alias name after `>` at the end of a line in order to tie domains together:

```
buythisnow.com bestdealever.com youneedthisstuff.com > scamsites
```

If you're using wildcard certificates, always define an alias name for it, so that you don't end up with a `*` character in your folder name:

```
*.foo.bar > wildcard.foo.bar
```

### Dehydrated HTTP-01 Challenge

In order to test Dehydrated with the HTTP-01 challenge, a web server serving a web site must be set up, say Apache 2 serving the site `foobar.com` (see *Appendix A*). Dehydrated will create the file needed to pass challenge according to the information provided by the CA upon request, and clean it up after the challenge succeeded.

For a web server serving its data from `/var/www`, create a directory `/var/www/acme` owned by `acme:acme`:

```
# mkdir /var/www/acme
# chown -R acme:acme /var/www/acme
```

This directory must be made available for every site whose certificates are going to be managed by Dehydrated under the path `/.well-known/acme-challenge`, both via HTTP and HTTPS. For the Apache web server, a reusable configuration can be created as follows (e.g. under `/etc/apache2/acme.config`):

```
Alias /.well-known/acme-challenge/ /var/www/acme/
<Directory "/var/www/acme/">
        Options         None
        Require         all granted
        AllowOverride   None
        ForceType       text/plain
</Directory>
```

For each virtual host supposed to serve TLS certificates managed by Dehydrated, add the following line to the configuration:

```
<VirtualHost *:443>
    Include /etc/apache2/acme.config
    ...
</VirtualHost>
```

#### Running Dehydrated

Make sure that the `hook.sh` script for the HTTP-01 challenge is available under `/etc/dehydrated/hook.sh`. When using another location, set the configuration option `HOOK` in `/etc/dehydrated/config` pointing to that script. Also set the `WELLKNOWN` option to `/var/www/acme`, so that the challenge files end up in that directory, which was setup before.

Dehydrated is now ready to run. Make sure you can run `dehydrated` either using `su` or `sudo`, depending on your setup:

```
$ su -m acme -c 'dehydrated -v'
$ sudo -u acme dehydrated -v
```

If this command's output includes version information about Dehydrated, everything is ready to run the registration command (`sudo` is used for all `dehydrated` commands henceforth):

```
$ sudo -u acme dehydrated --register --accept-terms
# INFO: Using main config file /etc/dehydrated/config
 + Generating account key...
 + Registering account key with ACME server...
 + Fetching account ID...
 + Done!
```

Dehydrated can now be run to request the certificates. Certificates that are missing, or that will will expire within the next 30 days, are requested by providing the `--cron` option:

```
$ sudo -u acme dehydrated --cron
# INFO: Using main config file /etc/dehydrated/config
Processing foobar.com
 + Signing domains...
 + Generating private key...
 + Generating signing request...
 + Requesting new certificate order from CA...
 + Received 1 authorizations URLs from the CA
 + Handling authorization for foobar.com
 + 1 pending challenge(s)
 + Deploying challenge tokens...
 + Responding to challenge for foobar.com authorization...
 + Challenge is valid!
 + Cleaning challenge tokens...
 + Requesting certificate...
 + Checking certificate...
 + Done!
 + Creating fullchain.pem...
 + Done!
```

This should output a list of domains for which certificates are going to be checked for expiration periodically. Since no certificates existed yet, they have been requested right away. The challenge files are cleaned up automatically.

### Deploying the Certificate

The certificate files end up in a sub-directory of `BASEDIR` (e.g. `/var/acme`):

- `accounts/` contains the account information resulting from the registration. There is one sub-directory for each CA account.
- `archive/` contains all the expired certificate, key, chain, and CSR files, which are kept around for later inspection.
- `chains/` containes cached certificate chain files, which are used to speed up the process of building new certificate chains.
- `cert/` contains the current certificate, key, chain, and CSR files. For each domain, a sub-directory named after its common name is created.

Every sub-directory of `cert/` contains the actual chain file to be deployed. The file name contains the epochal timestamp of the certificate creation time. Symlinks from the CSR (`cert.csr -> cert-1627207259.csr`), the certificate (`cert.pem -> cert-1627207259.pem`), the chain (`fullchain.pem -> fullchain-1627207259.pem`), and the private key (`privkey.pem -> privkey-1627207259.pem`) are created automatically, so that the paths to be used from the web server configuration remain stable. Use the paths to `fullchain.pem` and `privkey.pem` for your webserver configuration (`/etc/apache2/sites-enabled/foobar.com.conf`):

```
SSLEngine             on
SSLCertificateFile    /var/acme/certs/foobar.com/fullchain.pem
SSLCertificateKeyFile /var/acme/certs/foobar.com/privkey.pem
```

Make sure to restart your web server or to reload its config after modifying those paths:

```
# systemctl restart apache2.service
```

### Cleanup

Since renewed certificates end up in the same folder, old certificate, CSR, chain, and private key files should be archived once in a while:

```
$ sudo -u acme dehydrated --cleanup
```

The archived files, which end up in a sub-directory of `/var/acme/archive` (or generally speaking: in `${BASEDIR}/archive`), should be deleted once in a while using a cron job running a command as follows, which deletes archive files older than 300 days:

```
# find /var/acme/archive -type f -mtime 300 -delete
```

Notice that most web servers load the certificate files on startup and won't reload renewed certificate chains automatically. Consider running your web server's reload or restart command after certificate renewal using a cron job. Or as a better alternative, put this command into the `deploy_cert()` function of your `hook.sh` script. Make sure that the user running `dehydrated` has the according rights.

### Debugging

For debugging, make sure that the challenge files are created in the first place:

```
# watch -n 1 find -f /var/www/acme
```

This lists the contents of `/var/www/acme` every second, and a challenge token should appear while `dehydrated` is running. If not, something with your Dehydrated config or access rights to `/var/www/acme` must be wrong.

If the challenge file was created, but the challenge failed nonetheless, double check your Apache configuration; probably the challenge files aren't served.

## Dehydrated DNS-01 Challenge

Whereas the HTTP-01 challenge verifies that a web server requesting certificates can create arbitrary files in the directories served by it, the DNS-01 challenge verifies that a DNS server requesting certificates can create arbitrary `TXT` records for the domains managed by it.

Therefore, the DNS-01 challenge requires access to a domain's DNS configuration—and some familiarity with DNS on the side of the system's administrator. (*Appendix B* describes the process of setting up a basic authoritative-only DNS server which can be used to test the Dehydrated setup.)

For each SAN a certificate is to be managed by ACME, a `TXT` entry under a specific subdomain, say `_acme-challenge`, must be created; e.g. `_acme-challenge.foobar.com` for the domain `foobar.com`, or `_acme-challenge.www.foobar.com` for its subdomain `www.foobar.com`. According `CNAME` entries can be used to point to those `TXT` entries managed by the stub-DNS resolver used for ACME. Dehydrated must be able to update a `TXT` record on its own. When the `dehydrated` script runs, it creates one such `TXT` entry, which is used for all the challenges domains are pointing to using their `CNAME` record. This `TXT` record is deleted after the challenge succeeded (or failed).

### Dynamic Zone Setup

Since the `TXT` record is going to be managed dynamically, a dynamic DNS key needs bo be created using `ddns-confgen(8)`:

```
# ddns-confgen -k acme
```

Copy the following part of the output into `/etc/bind/acme.key`, and rename the key from `acme` to `acmekey`. Ignore the rest of the output:

```
key "acmekey" {
        algorithm hmac-sha256;
        secret "S6u5iSJYaur7K...";
};
```

In `/etc/bind/named.conf.local`, make sure to include that key:

```
include "/etc/bind/acme.key";
```

Also adjust the zone definition for `_acme-challenge.foobar.com` as created in *Appendix B*, so that it looks as follows:

```
zone "_acme-challenge.foobar.com" {
        type master;
        file "/var/cache/bind/db._acme-challenge.foobar.com";
        allow-query { any; };
        update-policy {
                grant acmekey name _acme-challenge.foobar.com TXT;
        };
};
```

This `update-policy` allows for dynamic DNS updates of `TXT` records within the zone `_acme-challenge.foobar.com` to the key `acme`.

Check your configuration, restart the `bind9` service, and run a test:

```
# named-checkconf
# named-checkzone _acme-challenge.foobar.com \
  /var/cache/bind/db._acme-challenge.foobar.com
# systemctl restart bind9.service
$ dig -t txt +short test._acme-challenge.foobar.com @localhost
"this is a test"
```

### Creating a Test TXT Record

Next, `nsupdate(1)` is used to create the `TXT` required by the DNS-01 challenge. The key created before is loaded using the `-k` option. (The commands after the `>` prompt are to be entered interactively; the `show` command is used to double-check the configuration change to be performed.)

```
# nsupdate -k /etc/bind/acme.key
> server localhost
> update add _acme-challenge.foobar.com 300 TXT HelloWorld
> show
Outgoing update query:
;; ->>HEADER<<- opcode: UPDATE, status: NOERROR, id:      0
;; flags:; ZONE: 0, PREREQ: 0, UPDATE: 0, ADDITIONAL: 0
;; UPDATE SECTION:
_acme-challenge.foobar.com. 300 IN    TXT     "HelloWorld"
> send
> quit
```

Check if the `TXT` record is served correctly when queried (both locally and remotely):

```
$ dig -t txt +short _acme-challenge.foobar.com @localhost
"HelloWorld"
$ dig -t txt +short _acme-challenge.foobar.com @ns1.foobar.com
"HelloWorld"
```

Since this is only a configuration test, and the ACME client will create a `TXT` entry with the challenge secret on its own, the record can be deleted again:

```
# nsupdate -k /etc/bind/acme.key
> server localhost
> update delete _acme-challenge.foobar.com TXT
> send
> quit
```

Now the `TXT` record should be gone (no output should be shown):

```
$ dig -t txt +short _acme-challenge.foobar.com @localhost
```

### Setting up DNS Aliases

In order to make use of the `TXT` entry being managed dynamically by Dehydrated, each domain needs an alias of their `_acme-challenge` subdomain to the challenge domain's `_acme-challenge` subdomain. For this purpose `CNAME` entries need to be created, e.g.:

```
_acme-challenge.my-website.com  _acme-challenge.foobar.com
_acme-challenge.whatever.com    _acme-challenge.foobar.com
_acme-challenge.www.foobar.com  _acme-challenge.foobar.com
```

Use low TTLs (`300` or `600` seconds), so that no old ACME challenges can interfere with newer ones. No `CNAME` entry is needed for the actual challenge domain.

### DNS-01 Hook Script

Download Dehydrated's sample DNS-01 script, which uses `nsupdate(1)`, and save it to `/etc/dehydrated/hook.sh`. Some adjustments are needed:

First, change the content of the `NSUPDATE` variable, so that it points to the correct key file:

```
NSUPDATE='nsupdate -k /etc/bind/acme.key'
```

Second, change the instructions under the `"deploy_cert"` branch, so that the apache configuration is updated:

```
"deploy_cert")
    sudo systemctl reload apache2.service
```

Make sure `acme` can perform this step by adding the following line to the sudoers file using `visudo(8)`:

```
acme ALL=(root) NOPASSWD: /etc/init.d/apache2 reload
```

Test if `acme` can actually reload Apache using `sudo`:

```
$ sudo -u acme bash
$ sudo /etc/init.d/apache2 reload
```

The `printf` statement for the `"deploy_challenge"` and `"clean_challenge"` challenge step cases ("hooks") use the variables and arguments provided in and to the script and should work as intended. The following arguments are provided to the script:

1. the challenge step (`deploy_challenge`, `clean_challenge`)
2. the domain name being challenged (e.g. `www.foobar.com`)
3. the filename (not used for DNS-01, only for HTTP-01)
4. the secret that must go into the `TXT` record

Adjust your `/etc/dehydrated/config` for the right challenge type:

```
CHALLENGETYPE="dns-01"
```

Also consider requesting a wildcard certificate by adjusting `/etc/dehydrated/domains.txt` as follows, which, after all, is the main reason for all the DNS hassle::

```
foobar.com *.foobar.com
```

Better first use the staging environment until certificate renewal is proofed to work. Finally, `dehydrated` can be executed:

```
$ sudo -u acme dehydrated --cron
# INFO: Using main config file /etc/dehydrated/config
Processing foobar.com with alternative names: *.foobar.com
 + Signing domains...
 + Generating private key...
 + Generating signing request...
 + Requesting new certificate order from CA...
 + Received 2 authorizations URLs from the CA
 + Handling authorization for foobar.com
 + Handling authorization for foobar.com
 + 2 pending challenge(s)
 + Deploying challenge tokens...
 + Responding to challenge for foobar.com authorization...
 + Challenge is valid!
 + Responding to challenge for foobar.com authorization...
 + Challenge is valid!
 + Cleaning challenge tokens...
 + Requesting certificate...
 + Checking certificate...
 + Done!
 + Creating fullchain.pem...
 + Done!
```

The certificates have been created, and Apache should have been reloaded.

### Dehydrated Configuration per Domain

If most of your domains work with the HTTP-01 challenge, but some special domains need DNS-01 because they need a wildcard certificate, the Dehydrated configuration can be created by domain to some extent. Set the `DOMAINS_D` option to point to a directory where those configurations are to be stored in `/etc/dehydrated/config`:

```
DOMAINS_D="/etc/dehydrated/domains.d"
```

Create one file per common name (the first entry per line in `domains.txt`), which contains the settings to be overwritten for that specific domain, e.g. `/etc/dehydrated/domains.d/quxbaz.com` for the domain `quxbaz.com` (here, the challenge type and the path to a special hook script are set):

```
CHALLENGETYPE="dns-01"
HOOK="/etc/dehydrated/dns01-hook.sh"
```

Notice that some options, e.g. `CA`, cannot be overwritten by a domain-specific configuration. The output of `dehydrated` will show that a special config is used for such a domain.

### Certificate Renewal

If the Dehydrated setup works, schedule a cron job so that renewals are attempted once a week. Certificates expiring within the next 30 days can be renewed, so running one renewal attempt once per week gives four attempts before the certificates are expired.

Make sure to create the cron job for the user `acme`:

```
$ sudo -u acme EDITOR=vi crontab -e
```

The following line configures a cron job that runs every sunday evening at 8 o'clock p.m. and attaches the command's output to `/var/log/dehydrated/renewal.log`:

```
0 20 * * 7 dehydrated --cron >/var/log/dehydrated/renewal.log 2>&1
```

Make sure to prepare the log folder as follows:

```
# mkdir /var/log/dehydrated
# chown -R acme:acme /var/log/dehydrated
```

As an alternative, use systemd's built-in logger:

```
0 20 * * 7 systemd-cat dehydrated --cron
```

Whose output can be viewed using:

```
$ sudo -u acme journalctl -u cron
```

# Chapter 8: HSTS and CAA

Even though TLS is omnipresent nowadays, websites and applications deploying proper TLS certificates can still be subject to attacks. There are additions to TLS leveraging other protocols to address those issues. Two of those are *HTTP Strict Transport Security* (HSTS) and *Certification Authority Authorization* (CAA).

## HTTP Strict Transport Security

An attacker performing a *downgrade attack* forces the client to use old and broken algorithms and/or TLS versions—or even to fall back to unencrypted, bare HTTP communication. This can be achieved using a *man-in-the-middle* attack, which redirects and captures the communication between client and server using an attacker's proxy. Even a server redirecting all HTTP traffic to HTTPS won't help, because the proxy still offers the client weak HTTPS or HTTP.

A website only serving HTTPS can use HSTS to inform the client that there's won't be served anything under plain, unencrypted HTTP. A client receiving this information will switch to HTTPS—and reject further wekly encrypted or entirely unencrypted communication.

Since the HSTS header is cached on the client side, no fallback to HTTP will be possible within the indicated caching period. HSTS also applies for all sites on a host, so an unencrypted subdomain is not possible, once a HSTS header for the same host is out.

### Deploying HSTS

HSTS is activated by issuing the `Strict-Transport-Security` header. Its `max-age` sub-value sets the header's caching duration in seconds. When deploying a completely new website, pick a high value from the start (e.g. `max-age=31536000` for the duration of one year). For an existing website, start low and increase the value progressively as confidence with testing grows and no negative feedback is reported.

The `includeSubDomains` header tells the client that HSTS not only applies to the main domain, but also to all of its sub-domains. Test this option together with a low `mag-age` duration for

existing deployments, so that you become aware of possible issues with sub-domains quickly, and without long-lasting issues for clients.

HSTS is only activated after the first response of a server reaches the client. Therefore, the domain is vulnerable to man-in-the-middle attacks for first-time visitors. The Chrome browser maintains a list of domains serving HSTS, so that HSTS can be applied for the first request of a client to a server, if that server is on the list. Other browser also use Chrome's list.

HSTS domains can be submitted to a web form. A minimum `max-age` of one year is required. Once registered, the `preload` sub-value of the `Strict-Transport-Security` header can be added.

There's also a removal form to get of the preload list. Notice that it can take a long time until this removal information reaches all the clients by the means of software updates. Think twice and test well before committing yourself to the HSTS preload list, which is a proprietary mechanism, after all.

### Certificate Authority Authorization

An attacker might get a valid certificate based on a leaked private key. By accident, you might deploy a valid certificate not compliant to regulations, because you picked the wrong CA or validation method.

A Certificate Authority Authorization DNS record (`CAA`) defines which CA is authorized to issue a certificate for the respective domain. For the `keyword` field, use `issue` or `issuewild` for specific or wildcard certificates, respectively. The `CAA` record's value contains the name of the CA that is allowed to issue a (wildcard) certificate for the respective domain. On entry per CA and domain is required. The CA's website should mention the exact name to be used for the `value` field.

This entries define that *Let's Encrypt* can issue certificates for the domain `foobar.com`, but only *SwissSign* is allowed to issue wildcard certificates for the same domain:

```
foobar.com  3600  IN  CAA  0  issue      "letsencrypt.org"
foobar.com  3600  IN  CAA  0  issuewild  "swisssign.com"
```

The value `;` (semicolon) indicates that no CA is allowed to issue a certificate for the respective domain, which is useful to prevent any CA respecting CAA entries from issuing wildcard certificates:

```
foobar.com  3600  IN  CAA  0  issue      "letsencrypt.org"
foobar.com  3600  IN  CAA  0  issuewild  ";"
```

The `CAA` record is checked when a certificate is issued by the CA. Check if your CA respects `CAA` records and make sure to follow your CA's documentation. CAA is still optional, but might become mandatory in the future.

# Chapter 9: TLS Testing and Certificate Analysis

Even though most applications deploying TLS can be tested manually with little effort, systematic testing is better performed using automated tools, of which many exist.

## Testing Server Configuration

SSL Labs is a free service to test your TLS server configuration. It performs a wide assessment covering many TLS features, and reports a grade from A (best) to F (worst). Using default configuration settings and allowing old protocol versions and algorithms can yield surprisingly low grades. The grade, however, will improve quickly as you follow the reported hints. Test regularly: a high grade detoriates if the configuration is not adjusted to ever-evolving standards.

Test SSL provides a shell script to perform similar kinds of tests like SSL Labs, but also works on servers not facing the Internet or using different protocols than HTTP(S). Your operating system may offer a package for `testssl.sh`. The script is written in Bash and relies on basic Unix utilities. A simple test against a web server can be run as follows:

```
$ testssl https://foobar.com
```

The output contains similar information as SSL Lab's report. Multiple operating systems and clients are simulated during the test.

To test servers running other applications than web servers, indicate the protocol using the `--starttls`, or, short, `-t` flag:

```
$ testssl --starttls imap mail.foobar.com:993
```

Cryptcheck offers a similar service like SSL Labs and Test SSL. Test them to figure out which one suits your needs best.

Bad SSL offers examples demonstrating how bad certificates or misconfigured TLS/SSL behave with your client.

### Testing Certificate Transparency

A public CA must keep records of which certificates it signs, and publish that information to public certificate logs, which then can be checked by auditors (*Certificate Transparency*). The CA also must published timely information on its revoked certificates.

Certificate Transparency is not only useful for auditors scrutinizing CAs, but also for domain owners that want to figure out which certificates have been issued and are deemed trustworthy for one's domain. For this purpose, services based on public certificate logs such as Certificate Search or the Google Transparency Report can be used. Make sure that only certificates can be found that were issued by CAs you really used!

A TLS certificate itself contains proof that it was submitted to a Certificate Transparency log. The CA, before sending you the requested certificate, submits a preliminary certificate to a Certificate Transparency log, which is then signed using a *Signed Certificate Timestamp* (SCT) by the log. That SCT is then copied into the real certificate being returned from the CA to the requestor. SCT is still optional, but certificates lacking one might be considered unsafe in the future.

## Chapter 10: Becoming a CA

Even though using a professionally maintained CA is usually the best option, sometimes you need to run your own internal CA.

A *Private Trust Anchor* is based on a self-signed certificate, which initially is not trusted by the clients, until you install the root certificate's public key on them. This effort is proportional to the number of clients, servers, and applications that are going to use your own CA.

Running a CA on a virtual machine is fine for testing and education, but for a professional setup, your CA should run on a server not facing the Internet. Building and running your own CA teaches you a lot about TLS and X.509.

While everything can be build with OpenSSL alone, consider commercial CA solutions for a professional setup, such as easy-rsa, XCA, Dogtag, FreeIPA, or EJBCA. There is also a lot of software capable of signing certificates you might already be running, such as Active Directory, Puppet, FreeNAS, Hashicorp Vault etc.

You can also deploy ACME internally using Boulder (the open source ACME server from Let's Encrypt) or step-ca. ACME also allows you to build your own CA for small, internal setups.

### CA Components

An OCSP responder is available as `openssl-ocsp(1)`, which should not be exposed to the Internet. Updates might break your setup, so prepare for debugging and research sessions when running your own CA using OpenSSL.

Signing certificates require databases of issued and revoked certificates. Such a database can be run using `openssl-ca(1)`, which, however, doesn't support locking, and therefore is not suited for usage of multiple simultaneous users.

You need one Certificate Revocation List per signing certificate, which must be made publicly available by a web server. (For internal usage, a web server only internally reachable is fine.)

You also need an OCSP responder with access to your certificate databases. In a professional setup, those Internet-facing services should run on a different machine than the CA itself.

### OpenSSL Root CA Configuration

CA files are usually put into `/root/CA` and must be only accessible by the `root` user. Modern CAs use both a root and an intermediary certificate. The configuration file `openssl.cnf` helps to keep them separated—and your certificates and keys organized.

Your root and intermediary certificate are going to need separate configurations. Put the root certificate under `/root/CA/root`, and the configuration under `/root/CA/root/openssl.cnf`. The `[ ca ]` section contains settings that apply to the `openssl ca` command:

```
[ ca ]
default_ca = CA_default
```

The CA is, thus, configured in a section called `CA_default`:

```
[ CA_default ]
dir = /root/CA/root
```

This defines the directory where the CA is going to be put in. The `dir` setting then can be referred from a variable called `$dir` for further configuration.

```
certs         = $dir/certs
new_certs_dir = $dir/newcerts
crl_dir       = $dir/crl
```

Thus, critical certificates, such as the root certificate, are put into `$dir/certs`, new certificates into `$dir/newcerts`, and the Certificate Revocation List into `$dir/crl`. Create those directories beforehand.

```
database = $dir/index.txt
serial   = $dir/serial
```

The database of signed certificates is stored under `$dir/index.txt`, and the certificate serial numbers under `$dir/serial`.

```
private_key = $dir/private/ca.key.pem
certificate = $dir/certs/ca.cert.pem
```

This defines the location of your private key and CA certificate. Create `$dir/private` beforehand, only accessible to `root`.

```
crlnumber        = $dir/crlnumber
crl              = $dir/crl/ca.crl.pem
crl_extensions   = crl_ext
default_crl_days = 30
```

Those options are needed for certificate revocation, which you, hopefully, won't ever need. The `crlnumber` file contains the next CRL number to be used. The `crl` option points to the current PEM-encoded CRL. X.509 extensions used for CRLs are listed in the `crl_ext` file. A CRL is good for `default_crl_days`, and must be renewed within that period, i.e. while still valid.

```
name_opt     = ca_default
cert_opt     = ca_default
default_days = 375
preserve     = no
policy       = policy_strict
```

The `ca_default` option points to modern settings to be used for the signing command. A certificate, by default, should expire within 375 days. With `preserve = no`, some obsolete backward compatibilities are deactivated. The `policy` setting poitns to another section named `policy_strict`.

A root certificate should only be allowed to sign intermediate certificates; root and intermediate certificates therefore must belong to the same organization. A strict policy is configured as follows:

```
[ policy_strict ]
countryName            = match
stateOrProvinceName    = match
localityName           = match
organizationName       = match
```

```
organizatoinalUnitName = optional
commonName             = supplied
emailAddress           = optional
```

Thus, country, state/province, and organization must be the same in the CSR as in the signing certificate. Organization unit and email are optional settings, and any value can be supplied as the (mandatory) common name.

Settings for generating CSRs are pre-defined in the `req` section:

```
[ req ]
default_bits       = 4096
distinguished_name = req_distinguished_name
string_mask        = utf8only
default_md         = sha256
x509_extensions    = v3_ca
prompt             = no
```

Those are settings commonly used for CSRs as of the year 2021. Never set `default_keyfile` to your private key location, which might overwrite your CA's key by an incomplete command accidentally run.

Both `req_distinguished_name` and `v3_ca` point to further sections of those names:

```
[ v3_ca ]
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints       = critical,CA:true
keyUsage               = critical,digitalSignature,cRLSign,keyCertSign
```

Particular certificates are identified by a unique hash code (`subjectKeyIdentifier`). Public keys used to create those certificates require a key id and an optional issuer indication; both pieces of information will end up in the certificate (`authorityKeyIdentifier`). Created certificates can be used to sign further certificates (`CA:true`), which *must* be respected by the client (marked as `critical`); so CA certificates can be created with this configuration (`basicConstraints`). The CA is allowed to sign certificates and its own CRL (`keyUsage`), which, again, the client *must* accept (`critical`).

The certificates signed by the intermediate certificate must not be allowed to sign other certificates by themselves. Therefore, `pathlen:0` must be added to the `basicConstraints` to an otherwise identical `v3_intermediate_ca` section:

```
[ v3_intermediate_ca ]
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints       = critical,CA:true,pathlen:0
keyUsage               = critical,digitalSignature,cRLSign,keyCertSign
```

Pre-defined options for new CSRs can be defined in the `req_distinguished_name` section, as pointed to from the `req` section further above:

```
[ req_distinguished_name ]
C  = CH
ST = Luzern
L  = Luzern
O  = Frickelbude
OU = Software Development
CN = CA Root Certificate
```

**Create the Root CA**

Once configuration and directory structure are ready, the serial numbers for certificates and CRL need to be initialized. A random value is used in a productive setting, but starting at 1000 is fine for a test environment:

```
# echo 1000 >/root/CA/root/serial
# echo 1000 >/root/CA/root/crlnumber
```

Also create an empty index file used as the database for signed certificates:

```
# touch /root/CA/root/index.txt
```

Now it's time to create the root certificate with a very long lifetime (7300 days, i.e. 20 years), based on the settings configured before:

```
# openssl req -config /root/CA/root/openssl.conf -newkey rsa \
              -keyout /root/CA/root/private/ca.key.pem \
              -x509 -days 7300 -extensions v3_ca \
              -out /root/CA/root/certs/ca.cert.pem
```

Make sure to use a strong passphrase, and store that passphrase somewhere safe. Check if the certificate was created according to the settings you configured:

```
$ openssl x509 -in /root/CA/root/certs/ca.cert.pem -noout -text
```

Install this private root certificate `ca.cert.pem` to the trusted certificate store of your clients. The root CA is ready now.

### OpenSSL Intermediate CA Configuration

The intermediate CA is created with the same directory structure as the root CA, and initialize the serial numbers:

```
# mkdir -p /root/CA/intermediate/{certs,crl,csr,newcerts,private}
# echo 1000 >/root/CA/intermediate/serial
# echo 1000 >/root/CA/intermediate/crl_number
# cp /root/CA/root/openssl.cnf /root/CA/intermediate/
```

The configuration file copied in the last step must be adjusted to the intermediate CA. First, modify the `dir` setting:

```
[ CA_default ]
dir = /root/CA/intermediate
```

Thanks to the relative paths based on `$dir` used elsewhere in the configuration, the other settings now point to the right paths. Add settings for algorithm and certificate lifetime, and adjust file names to the intermediate CA (still in the `CA_default` section):

```
default_md      = sha256
default_days    = 375
...
private_key     = $dir/private/intermediate.key.pem
certificate     = $dir/certs/intermediate.cert.pem
crl             = $dir/crl/intermediate.crl.pem
...
copy_extensions = copy
policy          = policy_loose
```

The `copy_extensions` setting instructs OpenSSL to copy any extension set in the CSR but not by the CA. The `policy` setting points to a section called `policy_loose`, so rename and reconfigure the copied `policy_strict` section, so that the intermediate CA can sign a wider range of CSRs:

```
[ policy_strict ]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizatoinalUnitName = optional
commonName              = supplied
emailAddress            = optional
```

Only a Common Name must be supplied. In the `req_distinguished_name` section, the Common Name must be adjusted, stating that it is an intermediate, not a root CA:

```
[ req_distinguished_name ]
C  = CH
ST = Luzern
L  = Luzern
O  = Frickelbude
OU = Software Development
CN = CA Intermediate Certificate 1
```

The number 1 is attached, so that multiple intermediate CAs can be created later. Keep the other settings identical as foor the root CA.

#### Create the Intermediate CA

The configuration is now ready to create the certificate for the intermediate CA, mostly similar to creating the root CA certificate:

```
# openssl req -config /root/CA/intermediate/openssl.cnf -newkey rsa \
             -keyout /root/CA/intermediate/private/intermediate.key.pem \
               -out /root/CA/intermediate/csr/intermediate.cert.csr
```

Use a different but equally strong passphrase as for the root CA. The intermediate CA's CSR can now be signed using the root certificate and the password for its private key:

```
# openssl ca -batch -cofig /root/CA/root/openssln.cnf \
             -extensions v3_intermediate_ca -days 3600 -notext \
             -in /root/CA/intermediate/csr/intermediate.cert.csr \
             -out /root/CA/intermediate/certs/intermediate.cert.pem
```

A shorter but still quite long lifetime (10 years) is used for the intermediate CA certificate.

Check the database (`/root/CA/root/index.txt`), which should now contain one entry starting with `V` for "valid" (as opposed to `R` for "revoked", or `E` for "expired"). The other fields show the expiration date timestamp, the revocation date timestamp (missing for certificates not revoked yet), the serial number (starting at 1000), the file name (always `unknown` in this setup), and, finally, the Distinguished Name.

TODO: example

The database is updated as you run the `openssl ca` command in context of your root CA. A copy of the certificate is stored under the `newcerts` directory, named after its serial number. Make backups when signing certificates.

Copy the intermediate certificate as the foundation for chain files:

```
# cp /root/CA/intermediate/certs/intermediate.cert.pem /root/CA/chain.pem
```

### OCSP Responder Certificate

Your CA needs an OCSP responder, so that clients can check the revocation status of individual certificates signed by your CA. The OCSP responder needs a certificate itself, which must be signed by the intermediate CA. This certificate can be configured as follows under `/root/CA/intermediate/ocsp.cnf`:

```
[ req ]
prompt             = no
default_bits       = 4096
distinguished_name = req_distinguished_name
default_md         = sha256
default_keyfile    = ocsp.privkey.pem

[ req_distinguished_name ]
C  = CH
ST = Luzern
L  = Luzern
O  = Frickelbude
OU = OCSP
CN = OCSP Responder
```

Only the Organizational Unit and the Common Name differ from the settings used before. The OCSP certificate then can be created as follows:

```
# openssl req -config /root/CA/intermediate/ocsp.cnf -newkey rsa \
              -out /root/CA/intermediate/csr/ocsp.cert.csr
```

Once more, use a strong and unique passphrase. Configure the X.509 extensions needed for OCSP directly in the intermediate CA's `openssl.cnf`:

```
[ ocsp ]
basicConstraints       = CA:FALSE
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid,issuer
keyUsage               = critical,digitalSignature
extendedKeyUsage       = critical,OCSPSigning
```

With this OCSP policy in place, and the CSR created before, the OCSP certificate can be signed as follows:

```
# openssl ca -batch -config /root/CA/intermediate/openssl.cnf \
             -extensions ocsp -notext \
             -in /root/CA/intermediate/csr/ocsp.cert.csr \
             -out /root/CA/intermediate/certs/ocsp.cert.pem
```

When prompted, use the intermediary CA's passphrase for signing. An OCSP responder using this certificate can now be set up, available under a domain like `ocsp.[yourdomain].[tld]`, usually running on port 80.

## Appendix A: Web Server Setup Using Apache 2

In order to setup and test Dehydrated for the domain `foobar.com`, a web server must be running, serving that particular site. (Use a real domain owned by you instead.) This quick tutorial describes how to do so under Debian 10 (Buster).

First, install Apache 2:

```
# apt install apache2
```

Second, create the directory to serve your site from (replace `foobar.com` by your proper domain) with proper permissions:

```
# mkdir -p /var/www/foobar.com/public_html
# chown -R 755 /var/www/foobar.com/public_html
```

Third, create a simple test index page (`/var/www/foobar.com/public_html/index.html`):

```
<h1>Hello, World!</h1>
```

Fourth, create a configuration file both serving HTTP and (yet bogus) HTTPS (`/etc/apache2/sites-available/foobar.com.conf`):

```
<VirtualHost *:443>
    ServerAdmin  webmaster@foobar.com
    ServerName   foobar.com
    ServerAlias  www.foobar.com www2.foobar.com
    DocumentRoot /var/www/foobar.com/public_html

    ErrorLog  ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

    SSLEngine             on
    SSLCertificateFile    /etc/ssl/certs/ssl-cert-snakeoil.pem
    SSLCertificateKeyFile /etc/ssl/private/ssl-cert-snakeoil.key
</VirtualHost>
<VirtualHost *:80>
    ServerAdmin  webmaster@foobar.com
    ServerName   foobar.com
    ServerAlias  www.foobar.com www2.foobar.com
    DocumentRoot /var/www/foobar.com/public_html

    ErrorLog  ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Fifth, disable the default page and activate `foobar.com`. Also enable Apache's SSL module:

```
# a2dissite 000-default.conf
# a2ensite foobar.com.conf
# a2enmod ssl
```

Finally, the webserver can be restarted:

```
# systemctl restart apache2.service
```

If everything works, the demo page should be available under both HTTP and HTTPS:

```
$ curl http://foobar.com/index.html
$ curl -k https://foobar.com/index.html
```

## Appendix B: DNS Server Setup Using Bind9

If you want to test the DNS-01 challenge with Dehydrated, besides a web server, you also must run your own authoritative-only DNS server. Again the domain `foobar.com` is used as a placeholder, to be replaced with your real domain name. As an IP address, the placeholder `123.45.67.89` is used. Debian 10 (Buster) is used in this tutorial, too. This setup requires that both the DNS and web server are running on the same host.

For this setup, the DNS server only manages the `TXT` records needed for the ACME challenge. `CNAME` records are created on the main DNS server of your hosting provider, one per SAN, pointing to the `TXT` records managed on your ACME-only DNS server.

First, a `NS` record is needed, which assigns your subdomain (`_acme-challenge.foobar.com`) to the DNS server that will manage it (`ns1.foobar.com`). Use your registrar's web interface to create one.

Second, an `A` record is needed setting `ns1.foobar.com` to the IP address `123.45.67.89`. (Consider a glue record for this purpose.)

Make sure that UDP port 53 is open on your server:

```
$ nmap 123.45.67.89 -p 53
```

Install the Bind 9 on your server and check the version number:

```
# apt install bind9 bind9utils bind9-doc
# named -v
```

Next, enable and start the `bind9` service and check its status (the last line should read: "server is up and running"):

```
# systemctl enable --now bind9
# rndc status
```

Also enable the `bind9-resolvconf` service so that Bind can be used as the system's default DNS resolver:

```
# systemctl enable --now bind9-resolvconf
```

Check if `/etc/resolv.conf` uses `127.0.0.1` as its name server:

```
$ cat /etc/resolv.conf
nameserver 127.0.0.1
```

Deactivate zone transfer and activate the query log by adding the following options to `/etc/bind/named.conf.options`:

```
allow-transfer { none; };
querylog yes;
```

Define your zone in `/etc/bind/named.conf.local` as follows:

```
zone "_acme-challenge.foobar.com" {
        type master;
        file "/var/cache/bind/db.acme.foobar.com";
        allow-query { any; };
};
```

The path `/var/cache/bind` is used instead of `/etc/bind`, because the former is allowed using Debian's default App Armor settings.

To create the zone file `db._acme-challenge.foobar.com`, copy from the template:

```
# cp /etc/bind/db.empty /var/cache/bind/db._acme-challenge.foobar.com
```

And create a zone configuration as follows:

```
$TTL    300
$ORIGIN _acme-challenge.foobar.com.
@       IN      SOA     ns1.foobar.com. webmaster.foobar.com. (
                     2021072600         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                         2419200         ; Expire
                          86400 )       ; Negative Cache TTL

        IN      NS      ns1.foobar.com.

test    IN      TXT     "this is a test"
```

Check the global and zone configuration and restart the `bind9` service:

```
# named-checkconf
# named-checkzone _acme-challenge.foobar.com \
  /var/cache/bind/db._acme-challenge.foobar.com
zone _acme-challenge.foobar.com/IN: loaded serial 2021072600
OK
# systemctl restart bind9.service
```

Test your DNS setup on the server by querying the test `TXT` record defined above:

```
$ dig -t txt +short test._acme-challenge.foobar.com @localhost
"this is a test"
```

If this succeeds, also perform the same test from your local computer:

```
$ dig -t txt +short test._acme-challenge.foobar.com @ns1.foobar.com
"this is a test"
```

If this also works, your DNS server is ready.