

# most-trusted

🕒 Friday 4 April 2025 🔒 19 versions

## 1. Who do you TRUST most?

**Ziel:** Entwurf einer eigenen Root-CA mit vertrauenswürdigen SSL-Zertifikaten

Should you consider using self-signed certificate chains when free and widely recognized certificates are readily available?

- Self-signed certificates can be just as secure as publicly trusted certificates.
- They establish a chain of trust, where each party is well-known and directly controlled.
- You have full control of the PKI and the validity of each certificate. No external party can invalidate the certificates.

The whole foundation of Web Public Key Infrastructure (PKI) is built upon chains of trust: *"I trust this certificate as it has been issued by a reputable authority that I know I can trust."*

But for internal domains there is really no need to use a public issuer as the domains are accessed in a controlled environment. A good alternative is to roll your **own PKI** with a private Root Certificate Authority.

Also keep in mind that using a **public CA** for internal certificates will reveal more of your *internal infrastructure*, as details will be readily available on sites such as [crt.sh](https://crt.sh).

## 2. Motivation

Seit den "Snowden"-Enthüllungen im Jahr 2013 [🔗](#) ist klar, dass nur mit einer *echten* Ende-zu-Ende-Verschlüsselung eine wirklich vertrauliche Datenübertragung in jedem Netzwerk gewährleistet werden kann.

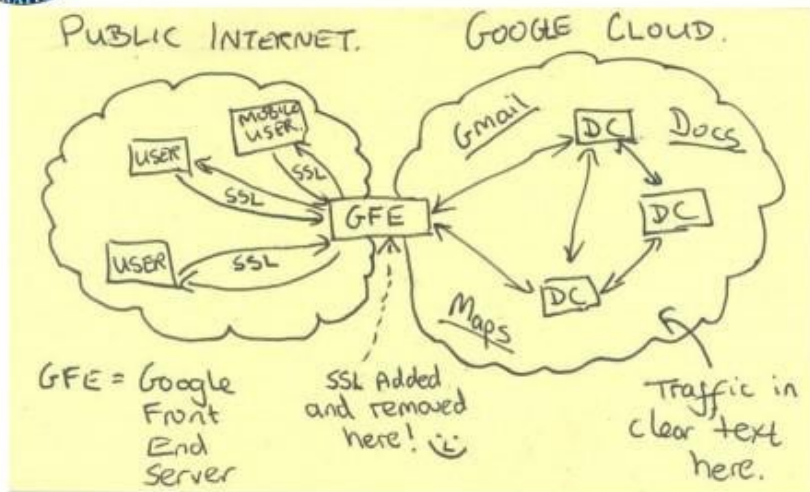
Dazu braucht man diverse Zutaten, wie eine **PKI**, eine Hierarchie von **CAs** und **X.509**-Zertifikate die **Chains-of-Trust** bilden. Dieser Vortrag soll diese Themen näher beleuchten

und die praktische Machbarkeit mit bekannten und neuen Open-Source-Tools (z.B. `cfssl` oder `smallstep`) zeigen.

TOP SECRET//SI//NOFORN



## Current Efforts - Google



TOP SECRET//SI//NOFORN

This 2013 National Security Agency (NSA) slide describing how data from Google's internal network was collected by intelligence agencies was eye-opening—and shocking—to many technology companies. The idea that an attacker could read messages passed between services wasn't technically groundbreaking, but it did reveal a security flaw in the way many distributed systems were designed. Many companies only encrypted the data to the border of their datacenter, not inside. The slide showed that private physical networks are being subverted to extract data passing through them. And just because a network has a [security perimeter](#), it doesn't mean that data can be safely sent between applications unencrypted inside that perimeter. In short: treat your own network as hostile.

### 3. Definitionen

Eine "Public Key Infrastructure" (PKI) ist ein System aus vertrauenswürdigen Zertifizierungsstellen (**CAs**), die eine hierarchische Vertrauenskette (Chain-of-Trust) bilden, um einem Endgerät und einem Netzwerkserver eine abgesicherte Kommunikation via "Transport Layer Security" (TLS) zu ermöglichen.

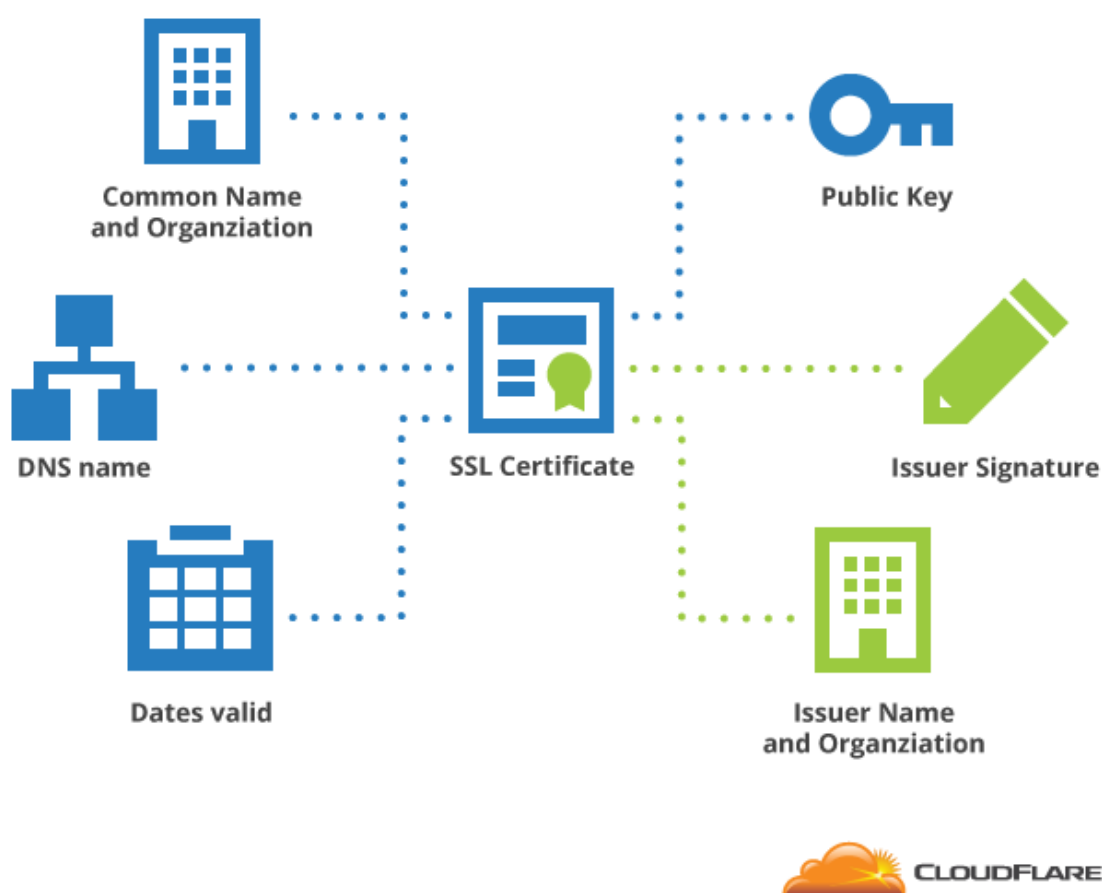
**TLS** (früher SSL) ist die Basis für das **HTTPS**-Protokoll. Es werden **X.509 Zertifikate** benutzt, um das Vertrauen zwischen den Kommunikationspartnern herzustellen und danach einen Session-Schlüssel auszutauschen und zu benutzen.

X.509-v3 ist ein ITU-Standard, der das Format von "Public Key Certificates" definiert. Ein X.509 Zertifikat verknüpft eine *Identität* mit einem *Public Key* mittels einer *Digital Signature*.

X.509 Zertifikate existieren in Base64 Formaten **PEM** (.pem, .crt, .ca-bundle), **PKCS-7** (.p7b, .p7s) und binären Formaten **DER** (.der, .cer), **PKCS-12** (.pfx, .p12).

Die technische Basis für die Erstellung der *Public Keys*, *Private Keys* und *Digital Signatures* ist die asymmetrische Verschlüsselungstechnik. Entweder unter Nutzung des **RSA**-Algorithmus oder auf Basis von **ECC** (Elliptic Curve Cryptography). Übersichten über beide Verfahren können [hier](#) oder [hier](#) eingesehen werden.

## The anatomy of a certificate



Im Blog von [Bityard](#) gibt's eine gute Zusammenfassung und Übersicht zum Thema.

### 4. Anleitung von SUSE

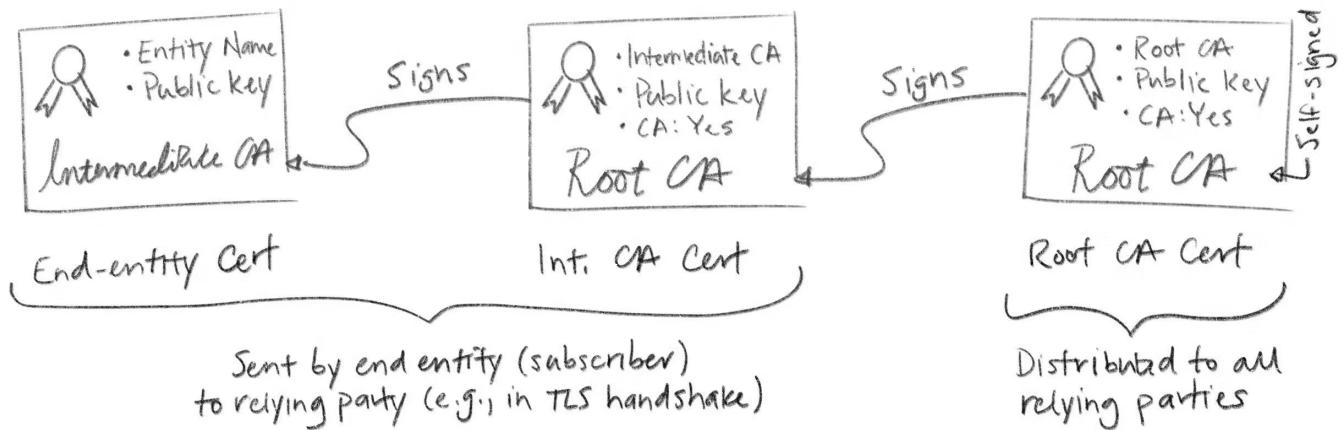
- Internet-Ressourcen: [Link](#) zum Artikel und [Glossar](#) der Fachbegriffe.
  - [PDF](#)-Datei zum Herunterladen.
  - Als Tool wird `openssl` benutzt.
- Erzeugen von Client Keys, Certificate Signing Requests (**CSR**) und einer privaten **CA**.

- Erzeugen von Zertifikaten und Installation von "Certificate Chains" im Linux Trust Store.

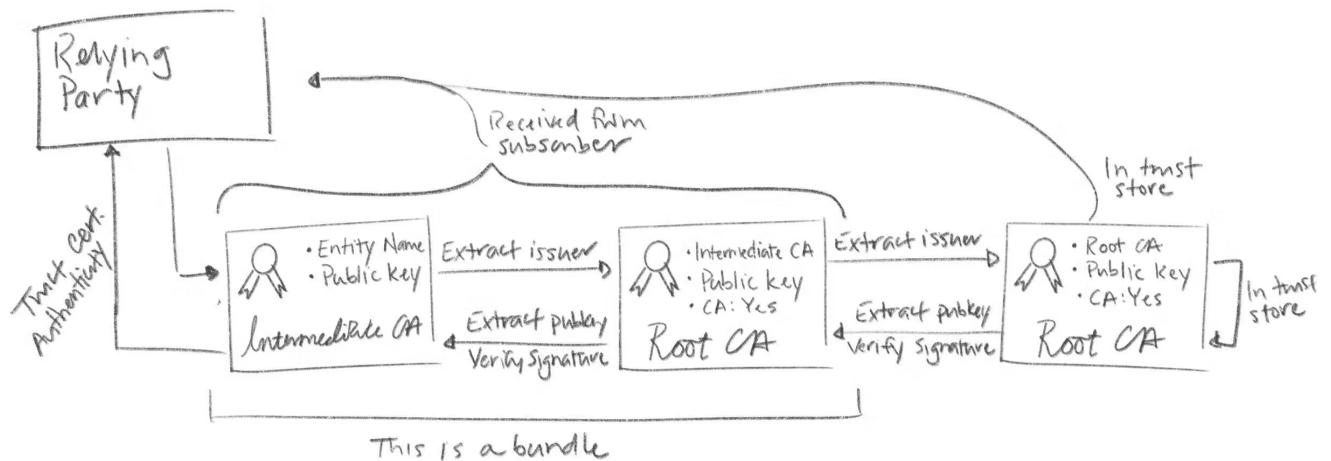
## 5. Methoden zum Aufbau einer eigenen Certificate Authority und Self-Signed Certificates

A certificate contains an *identity* (hostname, organization, etc.) and a *public key* (RSA, DSA, ECDSA, ed25519, etc.), and is either *signed* by a Certificate Authority or is Self-Signed.

### Certificate Chain



### Certificate Path Validation (simplified)



## 5.1. Werkzeuge für die Kommandozeile

### 5.1.1. DER Klassiker: `openssl`

Oft benutzt. Hat 'tausende' Optionen. → Kompliziert und unübersichtlich!

#### 5.1.1.1. Generate CA

---

##### 1. Generate RSA key

```
1 openssl genrsa -aes256 -out ca-key.pem 4096
```

##### 2. Generate a public CA Cert

```
1 openssl req -new -x509 -sha256 -days 365 -key ca-key.pem -out ca.pem
```

#### 5.1.1.2. View Certificate's content

---

```
1 openssl x509 -in ca.pem -text
2 openssl x509 -in ca.pem -purpose -noout -text
```

#### 5.1.1.3. Generate Certificate

---

##### 1. Create an RSA key

```
1 openssl genrsa -out cert-key.pem 4096
```

##### 2. Create a Certificate Signing Request (CSR): specify the *identity* as **subject** or *common name*.

```
1 openssl req -new -sha256 -subj "/CN=yourcn" -key cert-key.pem -out cert.csr
```

##### 3. Create an **extfile** with all the SANs (subject alternative names)

```
1 echo "subjectAltName=DNS:your-dns.record,IP:257.10.10.1" >> extfile.cnf
2 # optional
3 echo "extendedKeyUsage = serverAuth" >> extfile.cnf
```

##### 4. Let the CA create the signed certificate

```
1 openssl x509 -req -sha256 -days 365 -in cert.csr -CA ca.pem -CAkey ca-key.pem \
2 -out cert.pem -extfile extfile.cnf -CAcreateserial
```

#### 5.1.1.4. Convert Certificates

| COMMAND                                                           | CONVERSION |
|-------------------------------------------------------------------|------------|
| <code>openssl x509 -outform der -in cert.pem -out cert.der</code> | PEM to DER |
| <code>openssl x509 -inform der -in cert.der -out cert.pem</code>  | DER to PEM |
| <code>openssl pkcs12 -in cert.pfx -out cert.pem -nodes</code>     | PFX to PEM |

#### 5.1.1.5. Verify Certificates

```
1 openssl verify -CAfile ca.pem -verbose cert.pem
```

#### 5.1.2. Alternatives PKI-Tool von OpenVPN: `easy-rsa`

`Easy-rsa` ist ein [Shell-Skript](#) und Frontend für `openssl`.

- Tool und Quellcode können von [Github](#) heruntergeladen werden.
- Da ich keine Erfahrung damit habe, wird deshalb hier nur die Existenz erwähnt.

#### 5.1.3. `cfssl` von Cloudflare macht's einfacher

`cfssl` ist ein Tool von **Cloudflare** aus dem Jahr [2014](#), welches ohne die Verwendung von `openssl` auskommt.

- Syntax und Kommando-Optionen sind einfacher zu verstehen.
- Tool und Quellcode können von [Github](#) heruntergeladen werden.
- Ergänzende Blog-Artikel aus [2015](#) und [2016](#) erläutern weitere Details des Toolsets.

Da die offizielle Dokumentation von `cfssl` etwas dürftig ist, sind diese *privaten* Blog-Artikel sehr hilfreich!

- [Rob Blackburn](#)
- [Christofer J. Ekervhen](#)
- [Vianney Bouchaud](#)

- Yamli [↗](#)
- Kal Feher [↗](#)
- Srdan Dukic [↗](#)

Aus diesen Gründen ist **cfssl** das (bis jetzt) von mir bevorzugte Tool zum Arbeiten mit X.509-Zertifikaten. Diese Position könnte ihm aber von der *Neuentdeckung* **Smallstep** streitig gemacht werden.

### 5.1.3.1. Private-Key und CSR Dateien für einen Webservice erstellen

1. Spezifikation für den **CSR** als JSON-Datei erstellen. Dabei gleich alle **SANs** ("Subject Alternate Names") in den Abschnitt "hosts" einfügen. Das können sowohl DNS-Namen (FQDN) als auch IP-Adressen sein.

```

1 {
2   "CN": "my-service.fqdn.tld",
3   "key": {
4     "algo": "ecdsa",
5     "size": 256
6   },
7   "names": [
8     {
9       "C": "DE",
10      "ST": "RLP",
11      "L": "Ludwigshafen",
12      "O": "AG MicroComputer",
13      "OU": "Pagong"
14    }
15  ],
16  "hosts": [
17    "my-service.fqdn.tld",
18    "10.12.13.14"
19  ]
20 }
```

2. Public und Private **KEY** für den **CSR** erstellen und von JSON- in's PEM-Format wandeln.

```

1 $ cfssl genkey my-service-csr.json | cfssljson -bare my-service
2 2025/03/21 15:14:28 [INFO] generate received request
3 2025/03/21 15:14:28 [INFO] received CSR
4 2025/03/21 15:14:28 [INFO] generating key: rsa-2048
5 2025/03/21 15:14:28 [INFO] encoded CSR
6
7 $ ls -l
8 -rw-r--r-- 1 pagong users 1155 Mar 21 15:14 my-service.csr
```

```
9 -rw-r--r-- 1 pagong users 322 Mar 21 15:08 my-service-csr.json
10 -rw----- 1 pagong users 1679 Mar 21 15:14 my-service-key.pem
11
12 $ cfssl certinfo -csr my-service.csr
```

3. **CSR** Datei von (irgend)einer **CA** unterschreiben lassen. Damit erhält man die signierte **CRT** Datei.

- z.B. von `openssl` oder `step-ca` oder `OPNsense`
- oder auch von `cfssl` : siehe nächster Abschnitt

4. Fertiges Zertifikat (**CRT**-Datei) anzeigen.

```
1 $ cp my-service.pem my-service.crt
2 $ cfssl certinfo -cert my-service.crt
```

### 5.1.3.2. Signieren eines CSR von einer CA = Ausstellen eines Zertifikats (CRT) für einen Webservice

1. Lokales Signieren

```
1 $ cfssl gencert -ca my-ca.pem -ca-key my-ca-key.pem my-service-csr.json |
2     cfssljson -bare my-service
```

2. Remote Signieren

```
1 $ cfssl gencert -remote=remote_server my-service.csr.json |
2     cfssljson -bare my-service
```

3. Fertiges Zertifikat (**CRT**-Datei) anzeigen.

```
1 $ ls my-service*
2 my-service.csr
3 my-service-csr.json
4 my-service-key.pem
5 my-service.pem
6
7 $ cp my-service.pem my-service.crt
8 $ cfssl certinfo -cert my-service.crt
```

4. Beim `cfssl sign` Vorgang kann auch noch eine JSON-Konfigurationsdatei `cfssl-cfg.json` mit Profile-Definitionen genutzt werden, um das "Finetuning" des Verwendungszwecks (oder anderer Parameter) des Zertifikats anzupassen.

```
1 $ cfssl sign -config cfssl-cfg.json -profile profile_name \  
2   -ca ca.pem -ca-key ca-key.pem my-service.csr | \  
3   cfssljson -bare my-service
```

### 5.1.3.3. Eigene **Root-CA** erstellen

Dieses [Beispiel](#) wurde von Rob Blackburn übernommen.

1. Spezifikation für die **CA** als JSON-Datei erstellen.

```
1 {  
2   "CN": "Custom Widgets Root CA",  
3   "key": {  
4     "algo": "rsa",  
5     "size": 4096  
6   },  
7   "names": [  
8     {  
9       "C": "GB",  
10      "L": "London",  
11      "O": "Custom Widgets",  
12      "OU": "Custom Widgets Root CA",  
13      "ST": "England"  
14    }  
15  ]  
16 }
```

2. Public und Private **KEY** für die **CA** erstellen. Die **Root-CA** ist daran erkennbar, dass sie selbst signiert ist.  
(Also **Issuer** = *Owner* bzw. *Subject*).

```
1 $ cfssl genkey -initca root-ca-csr.json |  
2   cfssljson -bare root-ca  
3  
4 $ ls root-ca*  
5 root-ca.csr  
6 root-ca-csr.json  
7 root-ca-key.pem  
8 root-ca.pem
```

3. **PEM**- in **CRT**-Datei kopieren und Zertifikat anzeigen.

```
1 $ cp root-ca.pem root-ca.crt  
2 $ cfssl certinfo -cert root-ca.crt
```

#### 5.1.3.4. Eigene (intermediate) Branch-CA erstellen

Dieses [Beispiel](#) wurde ebenfalls von Rob Blackbourn übernommen.

1. Spezifikation für den **CSR** der **Branch-CA** als JSON-Datei erstellen.

```
1 {
2   "CN": "Custom Widgets Branch CA",
3   "key": {
4     "algo": "rsa",
5     "size": 2048
6   },
7   "names": [
8     {
9       "C": "GB",
10      "L": "London",
11      "O": "Custom Widgets",
12      "OU": "Custom Widgets Branch CA",
13      "ST": "England"
14    }
15  ],
16  "ca": {
17    "expiry": "42720h"
18  }
19 }
```

2. Public und Private **KEY** für den **CSR** der **Branch-CA** erstellen. Dabei entsteht auch eine nicht benötigte **PEM**-Datei.

```
1 $ cfssl gencert -initca branch-ca-csr.json |
2   cfssljson -bare branch-ca
```

3. Und den **CSR** gleich von der **Root-CA** signieren lassen. Dabei wird die 'unnötige' durch die 'richtige' **PEM**-Datei ersetzt.

```
1 $ cfssl sign -config cfssl-cfg.json -profile branch_ca \
2   -ca root-ca.pem -ca-key root-ca-key.pem branch-ca.csr |
3   cfssljson -bare branch-ca
4
5 $ ls branch-ca*
6 branch-ca.csr
7 branch-ca-csr.json
8 branch-ca-key.pem
9 branch-ca.pem
```

4. **PEM**- in **CRT**-Datei kopieren und Zertifikat anzeigen.

```
1 $ cp branch-ca.pem branch-ca.crt
2 $ cfssl certinfo -cert branch-ca.crt
```

### 5.1.3.5. Eigene Leaf-CA erstellen

Dies erfolgt ähnlich zum Vorgang der Signierung der Branch-CA durch die [Root-CA](#). Nur wird in diesem Fall die **Leaf-CA** durch die **Branch-CA** signiert.

1. Spezifikation für den **CSR** der **Leaf-CA** als JSON-Datei erstellen.

```
1 {
2   "CN": "Custom Widgets Leaf CA",
3   "key": {
4     "algo": "ecdsa",
5     "size": 384
6   },
7   "names": [
8     {
9       "C": "GB",
10      "L": "London",
11      "O": "Custom Widgets",
12      "OU": "Custom Widgets Leaf CA",
13      "ST": "England"
14    }
15  ],
16  "ca": {
17    "expiry": "8760h"
18  }
19 }
```

2. Public und Private **KEY** für den **CSR** der **Leaf-CA** erstellen. Dabei entsteht auch eine nicht benötigte **PEM**-Datei.

```
1 $ cfssl gencert -initca leaf-ca-csr.json |
2   cfssljson -bare leaf-ca
```

3. Und den **CSR** gleich von der **Branch-CA** signieren lassen. Dabei wird die 'unnötige' durch die '*richtige*' **PEM**-Datei ersetzt.

```
1 $ cfssl sign -config cfssl-cfg.json -profile leaf_ca \
2   -ca branch-ca.pem -ca-key branch-ca-key.pem leaf-ca.csr |
3   cfssljson -bare leaf-ca
4
5 $ ls leaf-ca*
6 leaf-ca.csr
```

```
7 leaf-ca-csr.json
8 leaf-ca-key.pem
9 leaf-ca.pem
```

4. **PEM**- in **CRT**-Datei kopieren und Zertifikat anzeigen.

```
1 $ cp leaf-ca.pem leaf-ca.crt
2 $ cfssl certinfo -cert leaf-ca.crt
```

### 5.1.3.6. Verwendung von KEY und CRT Dateien

Umwandeln von **KEY**- und **CRT**-Dateien zu Kubernetes **secrets** zur Verwendung in einem mit TLS abgesicherten Web-Service.

```
1 $ CRT="$SRCDIR/my-service.crt"
2 $ KEY="$SRCDIR/my-service.key"
3 $ kubectl -n my-namespace create secret tls my-service-tls --cert="$CRT" --key="$KEY"
4 $ kubectl -n my-namespace describe secret my-service-tls
```

## 5.2. Grafische Werkzeuge

### 5.2.1. Firewall OPNsense (GUI im Web-Browser)

**OPNsense** ist eine **Firewall** -Software, die auf **FreeBSD** basiert. Neben der Aufgabe als Firewall können damit über Plugins auch viele andere Netzwerk-Dienste (u.A. DNS, DHCP, NTP, VPN, etc.) eingerichtet und verwaltet werden.

- Download von **OPNsense** als **ISO-Datei**
- Link zur **Online-Dokumentation**

**OPNsense** enthält Plugins, die ein grafisches Frontend zu **openssl** und **ACME** bieten:

- **HowTo "Setup Self-Signed Certificate Chains"**
  - etwas einfacher zu Bedienen als die Kommandozeile
  - aber ohne Hintergrundwissen zu X.509 bleibt's schwierig
- Online-Artikel zur Nutzung von **OPNsense** als **CA**:
  - **SSLTrust**
  - **Zenarmor**

- Online-Blog-Artikel zum Plugin `acme-client` :
    - [Dennis Schröder](#)
    - [ClouDNS](#)
    - [HomeNetworkGuy](#)
  - Links zur Online-Dokumentation von **OPNsense**:
    - Kapitel "Trust" [↗](#)
    - Details zum [Plugin-API](#) `acmeclient`
    - Details zum [Core-API](#) `trust`
- 

### 5.2.2. Windows

---

Im **MatrixPost**-Blog wird die Nutzung der PKI Möglichkeiten in Windows Server gezeigt.

| THEMA               | LINK                                     |
|---------------------|------------------------------------------|
| PKI Design          | <a href="#">Part 1</a> <a href="#">↗</a> |
| Offline Root CA     | <a href="#">Part 2</a> <a href="#">↗</a> |
| Intermediate CA     | <a href="#">Part 3</a> <a href="#">↗</a> |
| Client Certificates | <a href="#">Part 4</a> <a href="#">↗</a> |
| Auto Enrollment     | <a href="#">Part 5</a> <a href="#">↗</a> |

---

## 5.3. Automationswerkzeuge

---

### 5.3.1. Zertifikate für Kubernetes: `cert-manager`

---

Der `cert-manager` [↗](#) wird für das automatisierte Anfordern und Erneuern von Zertifikaten in Kubernetes-Clustern benutzt.

- Umfangreiche [Dokumentation](#) [↗](#) und Source-Code bei [Github](#) [↗](#) .
  - Installation erfolgt über ein [Helm-Chart](#) [↗](#) .
- 

### 5.3.2. Neuentdeckung: Open Source PKI `Smallstep`

---

Während der Vorbereitung dieses Vortrags bin ich auf ein “YouTube”-Video und [Blog](#) gestossen, in welchem Open-Source-Tools der Firma **Smallstep** zum Aufbau einer privaten **CA** benutzt wurden:

- [“Self-Hosted TRUST with your own Certificate Authority”](#) [↗](#)

Bei der weiteren Recherche habe ich dieses [Dokument](#) des Firmengründers **Mike Malone** gefunden. Dort werden alle Eigenschaften von X.509v3-Zertifikaten und deren Life-Cycle *klar und präzise* erläutert:

- [“Everything you should know about certificates and PKI but are too afraid to ask”](#) [↗](#)

Ich bin von **Smallstep** begeistert und werde mich verstärkt darum kümmern. Leider war die Zeit bis zum *LinuxTag* zu kurz, um tiefer einsteigen zu können. Die beiden u.g. Tools könnten `cfssl` ergänzen (oder sogar ersetzen).

#### 5.3.2.1. Step-CA

---

Server-Tool:

- [Dokumentation](#) [↗](#) ] zum Tool `step-ca` zum Betreiben einer Certificate Authority **CA**.
- Download des Source-Codes und der Software-Pakete von [Github](#) [↗](#)
- Betrieb als Linux-Daemon oder im Docker-Container möglich

#### 5.3.2.2. Step CLI

---

Client-Tool:

- [Dokumentation](#) [↗](#) zum CLI-Tool `step` zum Anfordern und Verwalten von Zertifikaten aller Art.
- Download des Source-Codes und der Software-Pakete von [Github](#) [↗](#)

#### 5.3.2.3. Beispiele

---

Diese Artikel im **Smallstep**-Blog werden als Basis und Inspiration für meine Experimente dienen.

- [“Run your own private CA & ACME server using step-ca”](#) [↗](#) (als PDF)
  - [“Build a Tiny Certificate Authority For Your Homelab”](#) [↗](#) (als PDF)
-

## 5.4. ACME: Automated Certificate Management Environment

### 5.4.1. Let's Encrypt

Das Projekt "[Let's Encrypt](#)" hat zum Ziel, dass **alle** Web-Seiten und **sämtlicher** Internet-Traffic nur noch in verschlüsselter Form (per **TLS**-Protokoll) übertragen werden.

- Früher waren die dafür benötigten Zertifikate recht teuer und konnten zusammen mit einer DNS-Domäne gekauft werden. (Achtung: **ABO**-Modell!)
- Es gibt aber DynDNS-Provider, die einer Hobby-Seite Zugang zu einer kostenlosen DNS-Domäne ermöglichen. Und seit es "Let's Encrypt" gibt (also seit [2015](#)), werden auch die dazu passenden X.509-Zertifikate kostenlos ausgestellt.

Zur Überprüfung, dass ein Webseiten-Betreiber tatsächlich die Kontrolle über den **FQDN** der Webseite hat, wurde das [ACME](#) -Protokoll eingeführt, welches "Domain-Validation" **DV** benutzt:

1. Auf Anfrage durch den **ACME**-Client erzeugt der **ACME**-Server eine zufällige Information, die der Client auf einer speziellen **HTTP**-Unterseite der Webseite (oder als TXT-Record im **DNS**-System) veröffentlicht.
2. Im nächsten Schritt überprüft der **ACME**-Server die **HTTP**-Unterseite (oder den TXT-Record). Falls dort die gleiche Zufallsinformation gefunden wird, die er vorher dem **ACME**-Client gesendet hat, kann er davon ausgehen, daß der Client **echt** ist. Mit anderen Worten: die Domäne ist validiert.
3. Zum Schluss stellt der **ACME**-Server ein zeitlich befristetes Zertifikat (meist 90 Tage) für die **DNS**-Domäne des **ACME**-Clients aus, welches für die Übertragung der per **TLS** verschlüsselten **HTTPS**-Seiten des Web-Servers genutzt wird.
4. Vor Ablauf der Frist muss sich der **ACME**-Client wieder an den **ACME**-Server wenden und um die Zertifikats-Verlängerung (d.h. Ausstellung eines 'frischen' Zertifikats) bitten.

Neben "Let's Encrypt" gibt es auch noch andere Provider, die die **ACME**-Protokolle implementiert haben und oft auch kostenlose X.509-Zertifikate ausstellen.

#### Neueste Nachrichten!

- Laut "Heise"-[Newsticker](#) arbeitet "Let's Encrypt" an der [Verkürzung](#) der Zertifikats-Gültigkeit von 90 auf **6 Tage**.
- Die Trump-Regierung [stoppt](#) die Förderung von Open-Source-Projekten, wie "Let's Encrypt", "TOR" und "F-Droid".

### 5.4.2. SmallStep ACME

---

Both `step` and `step-ca` are natively integrated with the ACME protocol. `step` can be used to request ACME certificates from any ACME server, while `step-ca` is a fully functional private ACME server that works with all popular ACME clients.

---

### 5.4.3. Zero-SSL

---

Im “Caddy” Webserver ist, anstelle von “Let’s Encrypt”, der Zertifikatsprovider “Zero-SSL” automatisch aktiviert.

---

### 5.4.4. Andere ACME-Clients

---

#### 5.4.4.1. Certbot

---

Certbot is EFF’s tool to obtain certs from “Let’s Encrypt”. It’s written in [Python](#). It can also act as a client for any other CA that uses the ACME protocol.

- [Website](#) 
- [Github](#) 

#### 5.4.4.2. LEGO

---

Let’s Encrypt/ACME client and library written in Go.

- [Website](#) 
- [Github](#) 

#### 5.4.4.3. acme.sh

---

A pure Unix shell script implementing ACME client protocol.

- [Wiki](#) 
  - [Github](#) 
- 
-



## 6. Eigene **Root-CA** in Betriebssystemen verankern

---

### 6.1. On SUSE

---

- Copy the CA certificate ( `My-CA.pem` ) or ( `My-CA.crt` ) to `/etc/pki/trust/anchors/`
- Now run:

```
1 sudo update-ca-certificates
```

Refer to documentation [here](#) and [here](#). See also Blog [post](#).

### 6.2. On Fedora

---

- Copy the CA certificate ( `My-CA.pem` ) to `/etc/pki/ca-trust/source/anchors/` or `/usr/share/pki/ca-trust-source/anchors/`
- Now run:

```
1 sudo update-ca-trust
```

Refer to documentation [here](#). See also Blog [post](#).

### 6.3. On Debian & Derivatives

---

- Copy the CA certificate ( `My-CA.pem` ) or ( `My-CA.pem` ) into `/usr/local/share/ca-certificates/` .
- Update the Cert Store with:

```
1 sudo update-ca-certificates
```

Refer to documentation [here](#)  and [here](#). 

### 6.4. On Arch

---

System-wide – Arch(p11-kit)  
(From arch wiki)

- Run (as root)

```
1 trust anchor --store My-CA.crt
```

- The certificate will be written to `/etc/ca-certificates/trust-source/My-CA.p11-kit` and the “legacy” directories automatically updated.
- If you get “no configured writable location” or a similar error, import the CA manually:
- Copy the certificate to the `/etc/ca-certificates/trust-source/anchors/` directory.
- and then

```
1 sudo update-ca-trust
```

wiki page [here](#) 

---

See also:

- [Hochschule Schmalkalden](#) 
- [MatrixPost](#) 

### 6.5. On Windows

---

Assuming the path to your generated CA certificate as `C:\My-CA.pem` , run:

```
1 Import-Certificate -FilePath "C:\My-CA.pem" -CertStoreLocation Cert:\LocalMachine\Root
```

- Set `-CertStoreLocation` to `Cert:\CurrentUser\Root` in case you want to trust certificates only for the logged in user.

OR

In Command Prompt, run:

```
1 certutil.exe -addstore root C:\My-CA.pem
```

- `certutil.exe` is a built-in tool (classic `System32` one) and adds a system-wide trust anchor.

Siehe auch "Step 4 – Import root certificate to users devices" im Kapitel "openssl" bzw. den "Step 5" im Kapitel "cfssl" in einer anderen [Fundstelle](#) [🔗](#).

For Windows:

```
1 mmc
```

- Click Open
- Go to File > Add/Remove Snap-in
- Click Certificates and Add
- Select Computer Account and click Next
- Select Local Computer then click Finish
- Click OK to go back to the MMC window
- Double-click Certificates (local computer) to expand the view
- Select Trusted Root Certification Authorities, right-click on Certificates, and select "All Tasks" then "Import"
- Click Next then Browse. Change the certificate extension dropdown next to the filename field to "All Files (\*.\*)" and locate the "*your-root-cert-name*.pem" file, click "Open", then "Next"
- Select "Place all certificates in the following store". "Trusted Root Certification Authorities store" is the default. Click "Next" then click "Finish" to complete the wizard
- To verify, see your imported root certificate if it's listed in the list of root certificates

Auch im "Proxmox"-Wiki gibt es einen Artikel zum [Hinzufügen](#) [🔗](#) von Zertifikaten zum Windows Trust-Store.

## 6.6. On macOS

---

Assuming the path to your generated CA certificate is `~/My-CA.pem`, run (as root):

```
1 security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain ~/My-CA.
```

A dialog box will appear asking for an administrator's username and password. Enter it, and it will be stored in the system keychain. This can be verified by opening the `Keychain Access` application ( `/Applications/Utilities/Keychain Access.app` ). On the sidebar under `System Keychains` select `System`, and the new certificate should be listed.

---

Siehe auch "Step 4 – Import root certificate to users devices" im Kapitel "openssl" bzw. den "Step 5" im Kapitel "cfssl" in einer anderen [Fundstelle](#) [↗](#).

For MAC OS:

- Open up Keychain Access by searching it on Spotlight (clicking the Magnifying glass located on the upper right corner)
  - Select File > Import Items. Find the generated root certificate. Select System as "Destination Keychain"
  - Enter password when prompted
  - Logoff to take effect
- 

Auch im "Proxmox"-Wiki gibt es einen Artikel zum [Hinzufügen](#) [↗](#) von Zertifikaten zum Trust-Store von macOS.

---

## 7. Eigene Root-CA in mobilen Geräten verankern

---

### 7.1. On Android

---

The exact steps vary device-to-device, but here is a generalised guide:

1. Open Phone Settings
2. Locate `Encryption and Credentials` section. It is generally found under `Settings > Security > Encryption and Credentials`
3. Choose `Install a certificate`
4. Choose `CA Certificate`
5. Locate the certificate file `My-CA.pem` on your SD Card/Internal Storage using the file manager.

6. Select to load it.
7. Done!

- 
- “Importing SSL Certificate on Android or iPhone” [TitanHq](#) 
- 

## 7.2. On iOS

---

Apple makes this far more difficult than it should be:

1. Send `My-CA.pem` to the iOS device through iCloud, AirDrop, or a direct download from your server.
2. After downloading a dialog will appear on screen telling you that the profile has been downloaded.
3. Open the Settings app, and a `Profile Downloaded` item will be at the top. If it is not there, you may find it in `General → VPN & Device Management`.
4. Click `Install`.
5. The device will ask for your passcode. Enter it.
6. The device will then warn you about the certificate. Click `Install` again.
7. And, as if clicking `Install` twice wasn't enough, a confirmation button will appear at the bottom of the screen. Click `Install` one last time.
8. Done!

- 
- “Importing SSL Certificate on Android or iPhone” [TitanHq](#) 
- 

## 8. Eigene Root-CA in Web-Browser integrieren

---

### 8.1. Browser: Firefox

---

Tipps von Mozilla:

1. “Set up Certificate Authorities (CAs) in Firefox” im Mozilla [Support](#)   
(enthält Tipps für Windows und macOS und Android und Linux)
2. You can do that in the Certificate Manager:  
“Tools > Options > Advanced > Certificates: View Certificates”

---

To import a client certificate to Firefox, open the menu and select Preferences.

Navigate to the Certificates tab and click View Certificates, then Your Certificates, and finally, Import.

Locate your PKCS12 file (.p12 or .pfx), enter the password, and confirm the import.

Your certificate will appear under Your Certificates in Firefox Certificate Manager.

- Im “Proxmox”-Wiki gibt es einen Artikel zum [Hinzufügen](#) von Zertifikaten zum Trust-Store von Firefox.
- “Howto import your certificate to the browser” der “World Intellectual Property Organization” [WIPO](#)
- “Import Certificate to Mozilla Firefox” [keyfactor](#)
- “Importing SSL Certificate in Mozilla Firefox” [TitanHq](#)

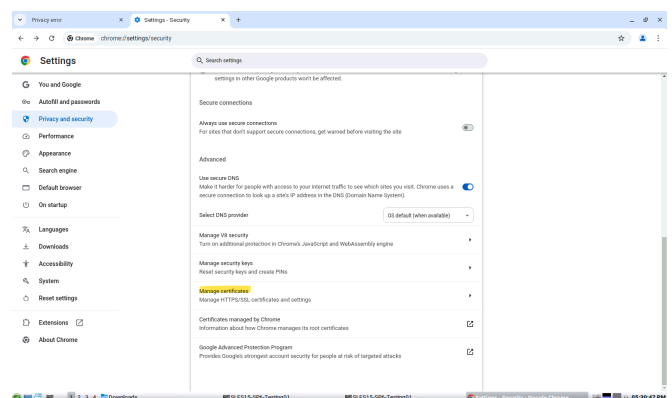
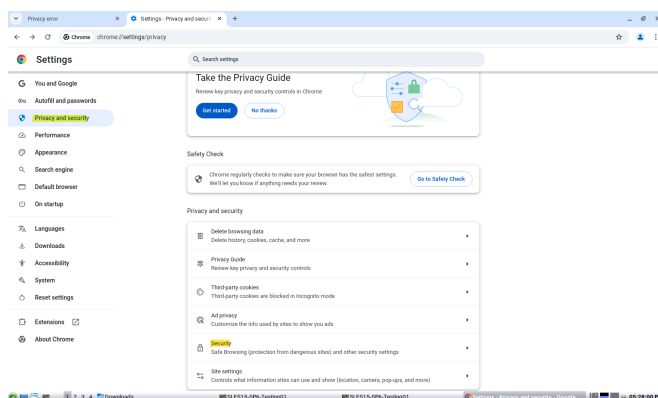
## 8.2. Browser: Chrome für Linux

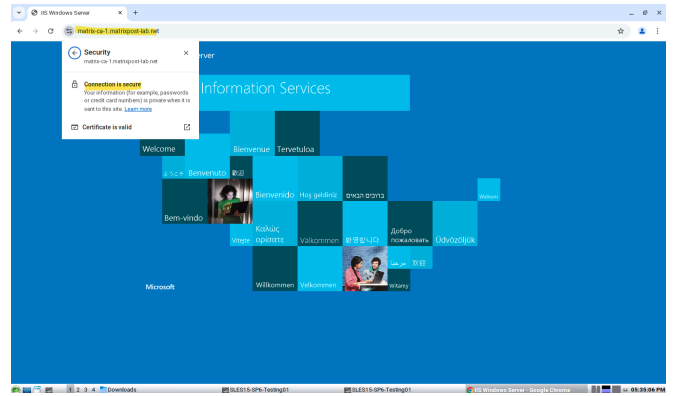
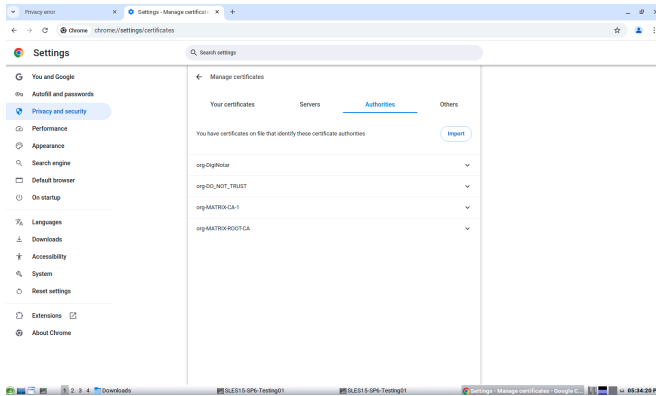
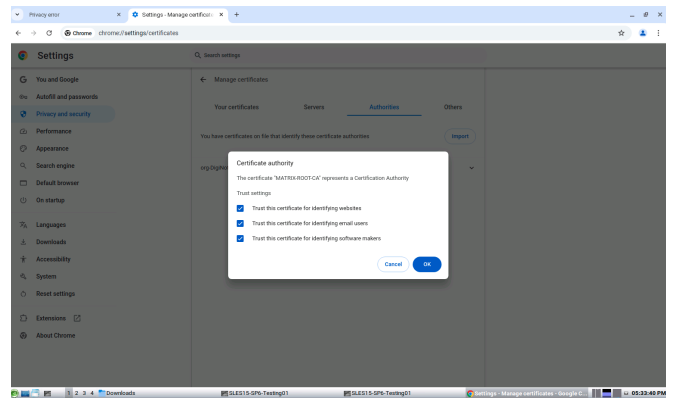
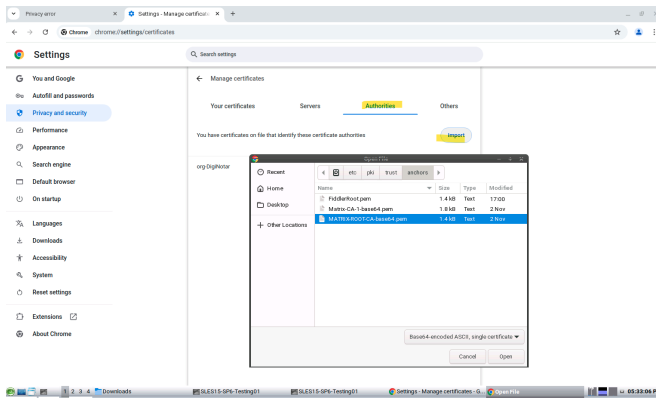
Viele Quellen sagen, dass “Chrome” (und “Edge”, etc.) den Trust-Store des Betriebssystems nutzen!

### 8.2.1. Install Root Certificates on Chrome running on Linux

For Linux, Chrome uses its own certificate store. You can import your Root CAs in Chrome directly.

In Chrome open *Settings* → *Privacy and security* → *Security* → *Manage certificates* → *Authorities*





Die Bilder stammen aus dieser guten [Anleitung](#) von **MatrixPost**!

- Im “Proxmox”-Wiki gibt es einen Artikel zum [Hinzufügen](#) von Zertifikaten zum Trust-Store von Chrome.
- “Importing SSL Certificate in Internet Explorer, Google Chrome or Opera” [TintanHq](#)

## 9. Backlinks

- [index](#) Tuesday 1 April 2025