Hitchhiker’s Guide To Certificates

Wolfgang Jung (post@wolfgang-jung.net)

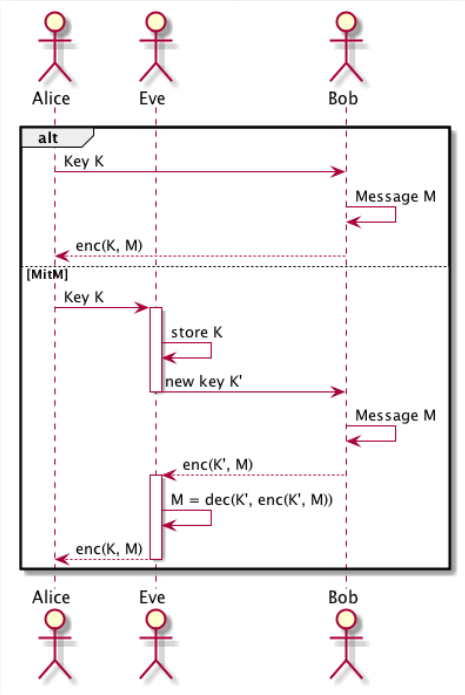
whoami

- einer der Micromata Gründer
- arbeite derzeit bei Polyas
- Schwerpunkte: Security, Infrastruktur, Linux, Scala

Sicherheit im Netz

- Verschlüsseln ist gut
- Aber: Woher den Schlüssel nehmen
- Symmetrisch
- Public Key/Private Key

MitM


Alice
Eve
Bob
alt
Key K
Message M
enc(K, M)
[MitM]
Key K
store K
new key K'
Message M
enc(K', M)
M = dec(K', enc(K', M))
enc(K, M)
Alice
Eve
Bob

Vertrauen aka "Wessen Schlüssel ist das?"

- Trust on first use
- DANE
- Certification Authorities

Trust on first use

- SSH nutzt dies
- Bei unbekanntem host-key: Frage den Nutzer
- Abgleich z.B. via

```
ssh -o VisualHostKey=yes localhost
Host key fingerprint is SHA256:stRf5rvgFGGRNEIdl9svIbJN8uHfzs0+TS8RzX4NjRw
+---[RSA 2048]----+
|       .oo=o..   |
|         .o+. E  |
|          o  + * |
|       . .o.= B +|
|      o S .Ooo *.|
|     . o ..=+ o *|
|      .   + .. =+|
|         o . .o+=|
|          . o. +B|
+----[SHA256]-----+
...
```

DANE

DNS-based Authentication of Named Entities

- Setzt DNSSEC voraus
- Für jeden Port Prüfsumme des Public Key hinterlegt:

```
_25._tcp.mail.ideas-in-logic.de. 3600   TLSA    3 1 1 \
  AD1730A7A5105E746EFFAA5DB6AE75A71B2B2BB48D506D9A44A270C9CEC0E928
```

- Prüfsumme reicht, da der Server ja sein Zertifikat dem Client mitschickt
- Email: Falls DANE-Record vorhanden ist, TLS zwingend
- Erweiterung für Emailadressen: rfc8162 (experimental)

CAs

Certification Authorities

- Externer Dienstleister, der Identität zu Public Key prüft
- Erzeugt ein Zertifikat für den Public Key
- Begrenzte Lebensdauer (aka Gelddruckmaschine)
- Begrenzter Einsatzzweck (aka Gelddruckmaschine)
- EV Zertifikate (aka Gelddruckmaschine)
- Kann Zertifikate zurückrufen (aka Pech gehabt)

Broken by design, but

- Jede CA ist immer vollständig gültig
- Certification Transparency (CT):
- Wenigstens grobe Verstöße werden sichtbar
- CAA (setzt DNSSEC voraus) begrenzt die Aussteller (Selbstverpflichtung)

Was ist denn nun ein Zertifikat?

- Basiert auf dem **Public-Key** des Inhabers
- Niemand außer des Inhabers sollte den **Private-Key** kennen
- Kann sowohl Client als auch Server betreffen!
- Distinguished Name des Ausstellers
- Distinguished Name des Inhabers
- Gültig von-bis
- Seriennummer (vergeben vom Aussteller)
- Public Key Verfahren (z.B. RSA)
- Public Key des Inhabers
- Signaturverfahren (z.B. sha256WithRSAEncryption)
- Erweiterungen: KeyUsage, Constraints, SAN, Revocation Lists, OCSP Responder, SCT (Signed Certificate Timestamp)

Woher weiß die CA, für wen sie ein Zertifikat ausstellt?

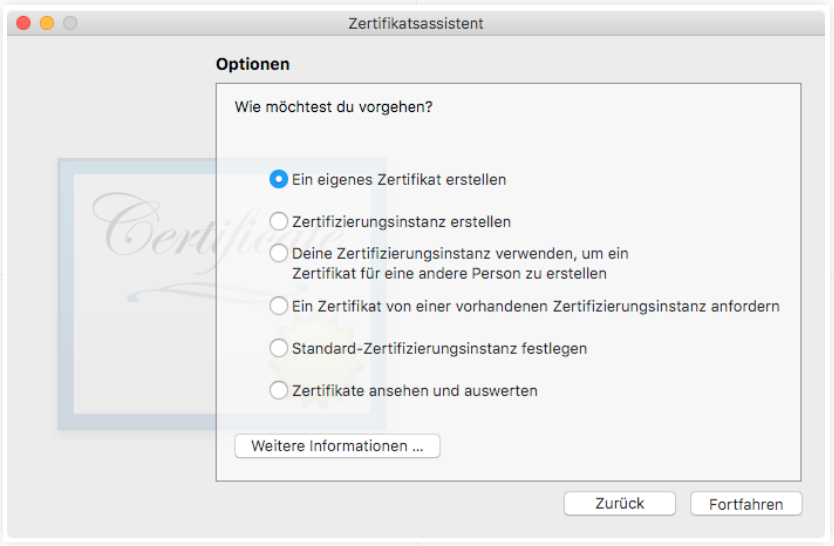
- Lösung CSR (Certificate Signing request)
- Enthält den X500 Namen des Inhabers
- Public Key Verfahren (z.B. RSA)
- Public Key des Inhabers
- Gewünschte Erweiterungen: SAN?, KeyUsage, etc.
- Signaturverfahren (z.B. sha256WithRSAEncryption)
- Signatur über diese Daten (via Private Key des Inhabers)
- Vorteil: CA kann Besitz des Private Keys prüfen, ohne ihn zu kennen

Intermediate CA

- Besondere Form der CA: wurde von CA signiert
- Vorteil: Nur Root-CA muss bekannt gemacht werden
- Intermediate CA kann/muss vom Server mitgesendet werden, da üblicheweise nicht auf dem Client bekannt
- Root-CA kann auf Airgap bzw. Hardware Modul liegen
- Intermediate-CA kann eingeschränkt werden z.B. nur E-Mail Zertifikate können ausgestellt werden

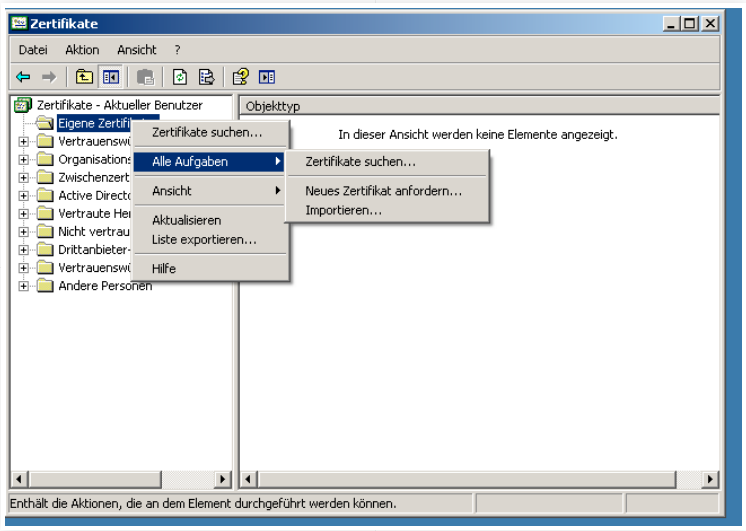
Wie erzeugt man nun Keys, CSRs etc? MacOS

- Klicki-Bunti: Schlüsselbund -> Zertifikatsassistent



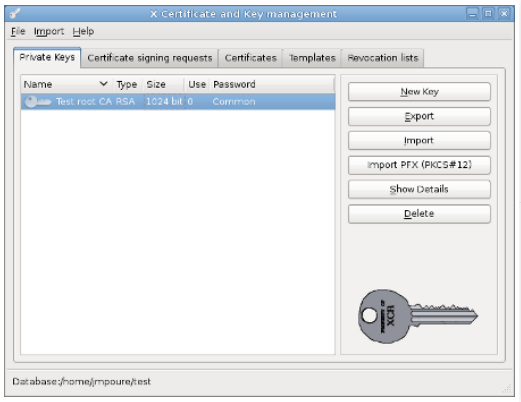
Windows

- certmgr.msc



Linux

- XCa


X Certificate and Key management
File Import Help
Private Keys
Certificate signing requests
Certificates
Templates
Revocation lists
Name
Type
Size
Use
Password
Test root CA
RSA
1024 bit
0
Common
New Key
Export
Import
Import PFX (PKCS#12)
Show Details
Delete
Database:/home/jmpoure/test

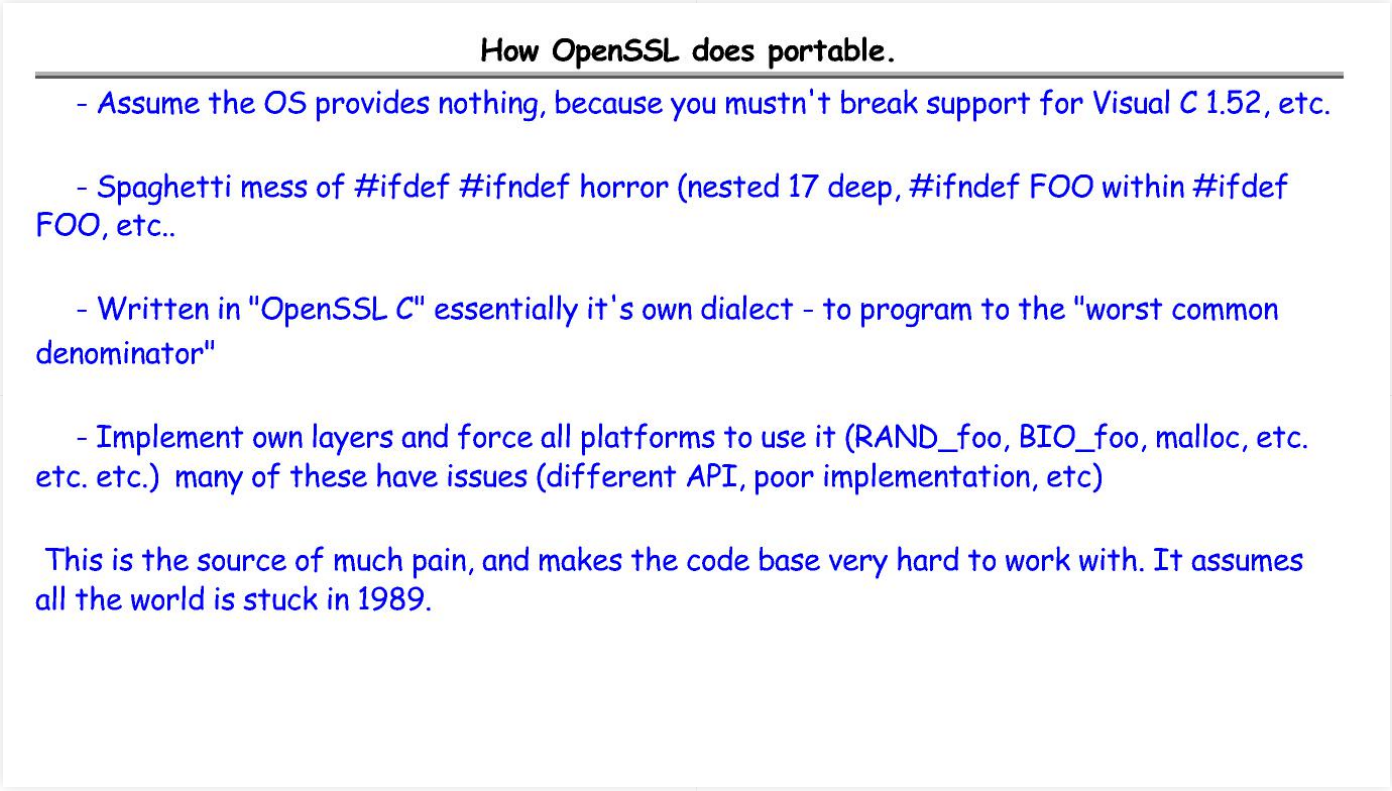
openssl CLI

- openssl help:

```
~ ❯ openssl help

Standard commands
asn1parse         ca                ciphers           cms
crl               crl2pkcs7         dgst*             dhparam
dsa               dsaparam          ec                ecparam
enc*              engine            errstr            exit
gendsa            genpkey           genrsa*           help
list              nseq              ocsp              passwd
pkcs12*           pkcs7             pkcs8*            pkey
pkeyparam         pkeyutl           prime             rand*
rehash            req*              rsa               rsautl
s_client*         s_server*         s_time            sess_id
smime             speed             spkac             srp
ts                verify*           version           x509*

Message Digest commands (see the `dgst' command for more details)
blake2b512        blake2s256        gost              md4
md5               mdc2              rmd160            sha1
sha224            sha256            sha384            sha512

Cipher commands (see the `enc' command for more details)
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
...
```

openssl?

- Erste Version 1995
- https://www.openbsd.org/papers/bsdcan14-libressl/:



How OpenSSL does portable.

- Assume the OS provides nothing, because you mustn't break support for Visual C 1.52, etc.

- Spaghetti mess of #ifdef #ifndef horror (nested 17 deep, #ifndef FOO within #ifdef FOO, etc..

- Written in "OpenSSL C" essentially it's own dialect - to program to the "worst common denominator"

- Implement own layers and force all platforms to use it (RAND_foo, BIO_foo, malloc, etc. etc. etc.) many of these have issues (different API, poor implementation, etc)

This is the source of much pain, and makes the code base very hard to work with. It assumes all the world is stuck in 1989.

Schlüssel erzeugen (RSA)

```
> openssl genrsa -out foo.key 4096
Generating RSA private key, 4096 bit long modulus
.....................................................................++
...................++
e is 65537 (0x010001)
```

Zertifikatsanforderung (CSR) erzeugen:

```
> openssl req -new -sha256 -key foo.key -out foo.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:Hesse
Locality Name (eg, city) []:Kassel
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Polyas GmbH
Organizational Unit Name (eg, section) []:Test
Common Name (e.g. server FQDN or YOUR name) []:localhost
...
```

CSR kann mittels `openssl req -text -in foo.csr` angeschaut werden.

CSR in Skript erzeugen

req.conf:

```
[req]
distinguished_name = req_distinguished_name
req_extensions = v3_req
prompt = no
[req_distinguished_name]
C = DE
ST = Hesse
L = Kassel
O = Polyas GmbH
OU = Tests
CN = localhost
[v3_req]
subjectAltName = @alt_names
[alt_names]
DNS.1=foobar
IP.1=1.2.3.4
```

```
openssl req -new -sha256 -key foo.key -out foo.csr -config req.conf
```

Formate für Zertifikate/Keys

- DER (Distinguished Encoding Rules): binär
- PEM (Privacy Enhanced Mail): base64
- Umwandlung z.B. von einem Zertifikat

```
openssl x509 -in foo.crt -outform der \
        -out foo.der
openssl x509 -in foo.der -inform der \
        -out foo.pem
```

Andere Formate

- PKCS#8: Enthält nur einen private Key

```
openssl pkcs8 -in foo.key  \
        -nocrypt -out foo.p8
```

- PKCS#12: Enthält private Key und Zertifikatskette

```
openssl pkcs12 -in foo.key  \
        -CAfile chain.pem   \
        -nodes -chain       \
        -out identity.p12
```

- "Raw" Public Key

```
openssl x509 -pubkey -noout -in foo.crt > pubkey.pem
```

Self signed certificate

Einfachste Form des Zertifikats:

```
openssl req -newkey rsa:4096 -nodes -keyout foo.key -x509 -days 365 -out foo.crt
```

Kurzform von:

```
openssl genrsa -out foo.key 4096
openssl req -new -sha256 -key foo.key -out foo.csr
openssl x509 -req -sha256 -days 365 -in foo.csr -signkey foo.key -out foo.crt
```

Zertifikat prüfen

```
openssl x509 -text -noout -in foo.crt
```

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            9e:05:fc:de:8c:b9:4d:01
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = DE, ST = Hesse, L = Kassel, O = Polyas GmbH, OU = foo, CN = foo
        Validity
            Not Before: Aug 13 23:43:22 2018 GMT
            Not After : Aug 20 23:43:22 2018 GMT
        Subject: C = DE, ST = Hesse, L = Kassel, O = Polyas GmbH, OU = foo, CN = foo
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:d0:24:19:48:ff:4b:61:2c:d3:42:bb:09:f8:6c:
...
                    34:c1:8b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage:
                Digital Signature, Key Encipherment, Data Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Subject Alternative Name:
                DNS:foo
    Signature Algorithm: sha256WithRSAEncryption
         bb:40:64:1b:1d:a9:09:48:d6:9e:90:34:c2:66:2d:06:5b:0c:
...
         c2:9c:25:21:a7:f7:9c:27
```

Eigene Root-CA?

- Auch nur ein self-signed Zertifikat

```
> openssl req -x509 -new -extensions v3_ca \
          -subj "/C=DE/ST=Hesse/L=Kassel/O=POLYAS GmbH/OU=CA/CN=POLYAS CA" \
          -key selfSignCA.key -days 10000 -out selfSignCA.crt
> openssl pkcs12 -export -inkey selfSignCA.key -in selfSignCA.crt \
          -out selfSignCA.p12
```

- üblicherweise dem Betriebssystem nicht bekannt
- Installation in Zertifikatsspeicher
- Oder: Aktivierung auf Applikationsebene z.B. play

```
play.ws.ssl {
  trustManager = {
    stores = [
      { type = "PEM", path = "conf/selfSignCA.crt" }
      { path: ${java.home}/lib/security/cacerts }
    ]
  }
}
```

Zertifikate eines Servers testen

```
> openssl s_client -connect google.de:443 -servername google.de -showcerts

CONNECTED(00000005)
depth=2 OU = GlobalSign Root CA - R2, O = GlobalSign, CN = GlobalSign
verify return:1
depth=1 C = US, O = Google Trust Services, CN = Google Internet Authority G3
verify return:1
depth=0 C = US, ST = California, L = Mountain View, O = Google LLC, CN = *.google.de
verify return:1
---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google LLC/CN=*.google.de
   i:/C=US/O=Google Trust Services/CN=Google Internet Authority G3
-----BEGIN CERTIFICATE-----
MIIEhzCCA2+gAwIBAgIITY5Z/D6ZRaQwDQYJKoZIhvcNAQELBQAwVDELMAkGA1UE
...
JGYmyIvgUsLO5Xo=
-----END CERTIFICATE-----
 1 s:/C=US/O=Google Trust Services/CN=Google Internet Authority G3
   i:/OU=GlobalSign Root CA - R2/O=GlobalSign/CN=GlobalSign
-----BEGIN CERTIFICATE-----
MIIEXDCCA0SgAwIBAgINAeOpMBz8cgY4P5pTHTANBgkqhkiG9w0BAQsFADBMMSAw
...
c7o835DLAFshEWfC7TIe3g==
-----END CERTIFICATE-----
---
Server certificate
subject=/C=US/ST=California/L=Mountain View/O=Google LLC/CN=*.google.de
issuer=/C=US/O=Google Trust Services/CN=Google Internet Authority G3
---
```

Serverzertifikate lokal testen

```
> openssl s_server -accept 9992 \
          -key w.jung@wolfgang-jung.net.key \
          -cert w.jung@wolfgang-jung.net.crt \
          -CAfile IdeasInLogicCA/selfSignCA.crt -WWW
> openssl s_client -connect pong:9992 -CAfile  IdeasInLogicCA/selfSignCA.crt

Using default temp DH parameters
ACCEPT


ACCEPT
140736242467776:error:1417C0C7:SSL routines:tls_process_client_certificate:peer did not return a certificate:ssl/statem/statem_srvr.c:2882:
```

Client-Auth

```
nginx.conf:
...
log_format combined_sslclient '$remote_addr - $remote_user [$time_local] "$request" \
           $status $body_bytes_sent "$http_referer" "$http_user_agent" \
           "$ssl_client_s_dn($ssl_client_serial)"';
...

site.conf:

server {
        ssl_certificate /etc/letsencrypt/live/wolfgang-jung.net/fullchain.pem;
        ssl_certificate_key /etc/letsencrypt/live/wolfgang-jung.net/privkey.pem;

        ssl_stapling on;
        ssl_stapling_verify on;
        ssl_trusted_certificate /etc/letsencrypt/live/wolfgang-jung.net/fullchain.pem;

        ssl_client_certificate /etc/nginx/client_certs/ca.crt;
        ssl_verify_client optional;

        listen 443 ssl http2;
        access_log /var/log/nginx/wolfgang-jung.net-access.log combined_sslclient;
        ...
}
```

Client-Auth

```
> openssl s_client -connect wolfgang-jung.net:443 -servername wolfgang-jung.net \
          -key w.jung@wolfgang-jung.net.key -cert w.jung@wolfgang-jung.net.crt \
          -CAfile  IdeasInLogicCA/selfSignCA.crt
```

```
CONNECTED(00000003)
depth=2 O = Digital Signature Trust Co., CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
verify return:1
depth=0 CN = wolfgang-jung.net
verify return:1
---
Certificate chain
 0 s:/CN=wolfgang-jung.net
   i:/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
 1 s:/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
   i:/O=Digital Signature Trust Co./CN=DST Root CA X3
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIGJjCCBQ6gAwIBAgISA8Uwxad/lgkqvfyWBVmRPzWbMA0GCSqGSIb3DQEBCwUA
...
FvDzUtL/Jc35A9cmdxth3ox/tdsxOMZZRfwDXEm1ubvSGW1oAP+cf36D
-----END CERTIFICATE-----
subject=/CN=wolfgang-jung.net
issuer=/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
---
Acceptable client certificate CA names
/C=DE/ST=Hesse/L=Kassel/O=Ideas In Logic GbR/OU=Local CA/CN=Ideas in Logic internal CA
Client Certificate Types: RSA sign, DSA sign, ECDSA sign
Requested Signature Algorithms: RSA+SHA512:DSA+SHA512:ECDSA+SHA512:RSA+SHA384:DSA+SHA384:ECDSA+SHA384:RSA+SHA256:DSA+SHA256:ECDSA+SHA256:RSA+SHA224:DSA+SHA224:ECDSA+SHA224:RS
Shared Requested Signature Algorithms: RSA+SHA512:DSA+SHA512:ECDSA+SHA512:RSA+SHA384:DSA+SHA384:ECDSA+SHA384:RSA+SHA256:DSA+SHA256:ECDSA+SHA256:RSA+SHA224:DSA+SHA224:ECDSA+SH
Peer signing digest: SHA512
Server Temp Key: ECDH, P-384, 384 bits
---
SSL handshake has read 5212 bytes and written 4077 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
```

access.log:

```
87.191.133.92 - - [15/Aug/2018:13:40:14 +0200] "GET / HTTP/2.0" 304 0 "-" \
                  "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) ...." \
                  "CN=w.jung@wolfgang-jung.net,OU=Wolfgang Jung,O=Wolfgang Jung,L=Kassel,ST=Hesse,C=DE(A179C4773E6065E0)"
```

Mutual SSL lokal testen

```
> openssl s_server -accept 9992 -Verify 1 \
          -key w.jung@wolfgang-jung.net.key \
          -cert w.jung@wolfgang-jung.net.crt \
          -CAfile IdeasInLogicCA/selfSignCA.crt -WWW
> openssl s_client -connect pong:9992 -key w.jung@wolfgang-jung.net.key \
          -cert w.jung@wolfgang-jung.net.crt -CAfile  IdeasInLogicCA/selfSignCA.crt

Using default temp DH parameters
ACCEPT
depth=1 C = DE, ST = Hesse, L = Kassel, O = Ideas In Logic GbR, OU = Local CA, CN = Ideas in Logic internal CA
verify return:1
depth=0 C = DE, ST = Hesse, L = Kassel, O = Wolfgang Jung, OU = Wolfgang Jung, CN = w.jung@wolfgang-jung.net
verify return:1

> openssl s_client -connect pong:9992 -CAfile  IdeasInLogicCA/selfSignCA.crt

ACCEPT
140736242467776:error:1417C0C7:SSL routines:tls_process_client_certificate:peer did not return a certificate:ssl/statem/statem_srvr.c:2882:
```

openssl was noch?

- Prüfsummen / Signaturen:

```
> openssl dgst -r -sha384 file
f861caf733f5dab899c6903a7f1aedd12bd4e4dde7c76e8afcf9e07cd984579dd668a60c9defee0a6490b32
> openssl dgst -r -sha384 -hmac MySecretKey file
b398e60376030fbe3f3998b9495ac21e32238ec85ed8158465adb001c9a8defa3bc2641feb8e67703f3b2f9
> openssl dgst -r -sha384 -sign foo.key -out sig file
> openssl dgst -r -sha384 -verify pubkey.pem -signature sig file
Verified OK
```

- Verschlüsselung:

```
> openssl aes-128-cbc -k SharedKey -e -a < file
U2FsdGVkX19wedpEWuvRd1kwUnrC6DCK5qkRqeuxffk=
> openssl aes-128-cbc -k SharedKey -d -a
Mein Text
```

- Zufall:

```
> openssl rand -hex 16
8ccd23881e30f1115b84c633c557f6c2
```

Fragen?