PKI Certificate

S B 2020

Agenda

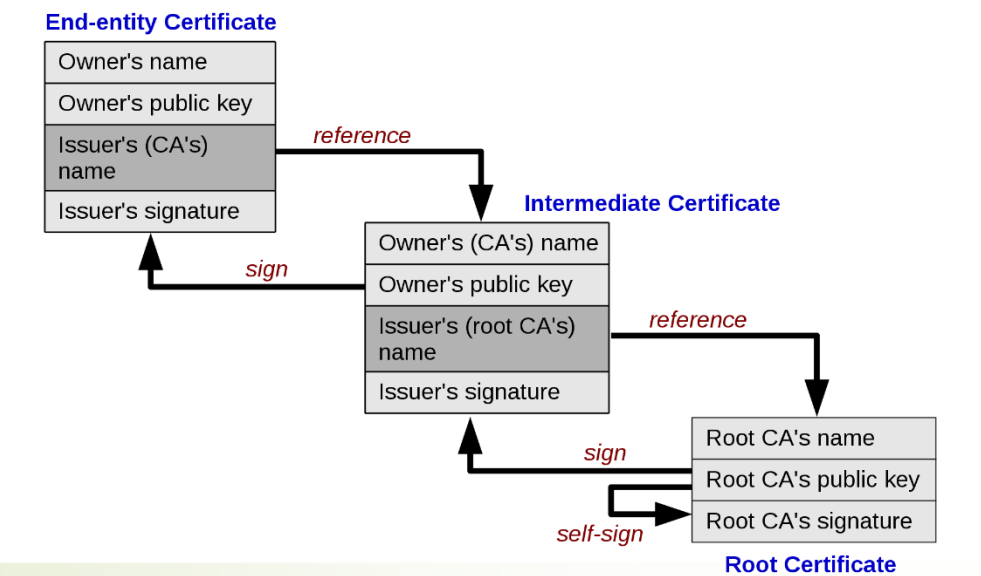
- Concept
- Usage
  - SSL over TCP/HTTP/LDAP
  - Encryption/Decryption
  - Identification
  - Code Signing
- How to
  - Check on the fly
  - Fill/fix broken certificate chain
  - Extract a Certificate on the fly
  - Create a new Certificate
  - Create s Self-signed Certificate

Concept

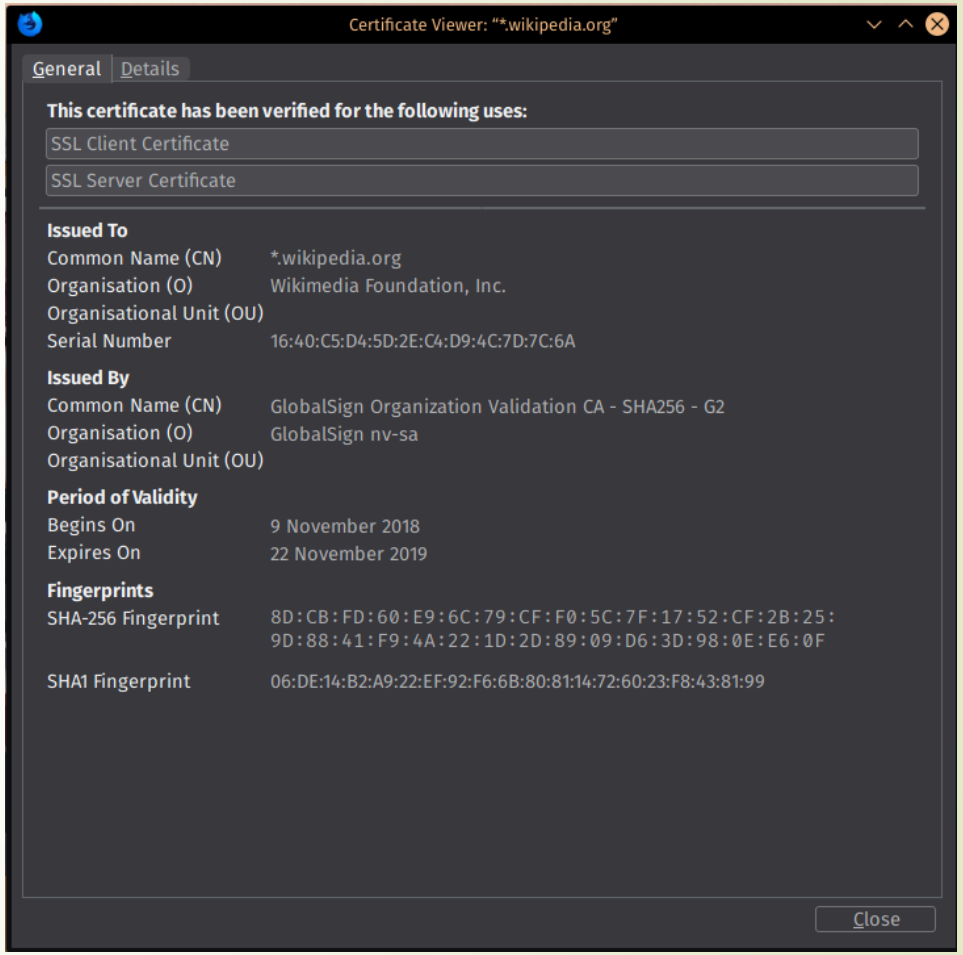
In cryptography, a **public key certificate,** also known as a **digital certificate** or **identity certificate,** is an electronic document used to prove the ownership of a public key.[1] The certificate includes information about the key, information about the identity of its owner (called the subject), and the digital signature of an entity that has verified the certificate's contents (called the issuer). If the signature is valid, and the software examining the certificate trusts the issuer, then it can use that key to communicate securely with the certificate's subject. In email encryption, code signing, and e-signature systems, a certificate's subject is typically a person or organization. However, in Transport Layer Security (TLS) a certificate's subject is typically a computer or other device, though TLS certificates may identify organizations or individuals in addition to their core role in identifying devices. TLS, sometimes called by its older name Secure Sockets Layer (SSL), is notable for being a part of HTTPS, a protocol for securely browsing the web.

Source: https://en.wikipedia.org/wiki/Public_key_certificate

Concept



The chain of trust



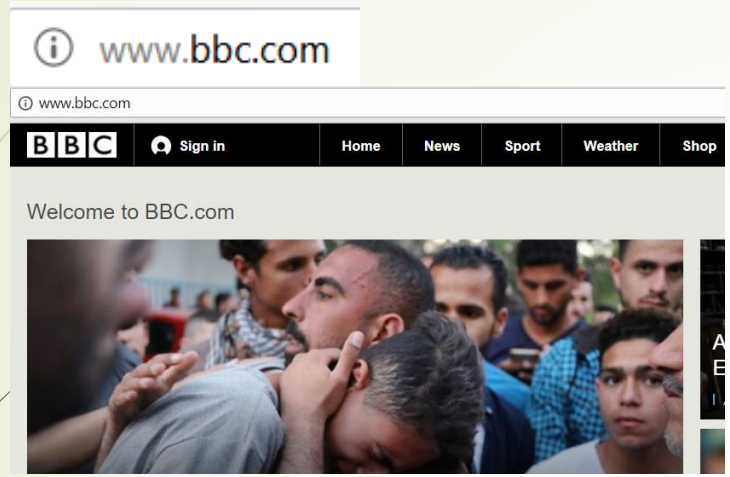
An example certificate

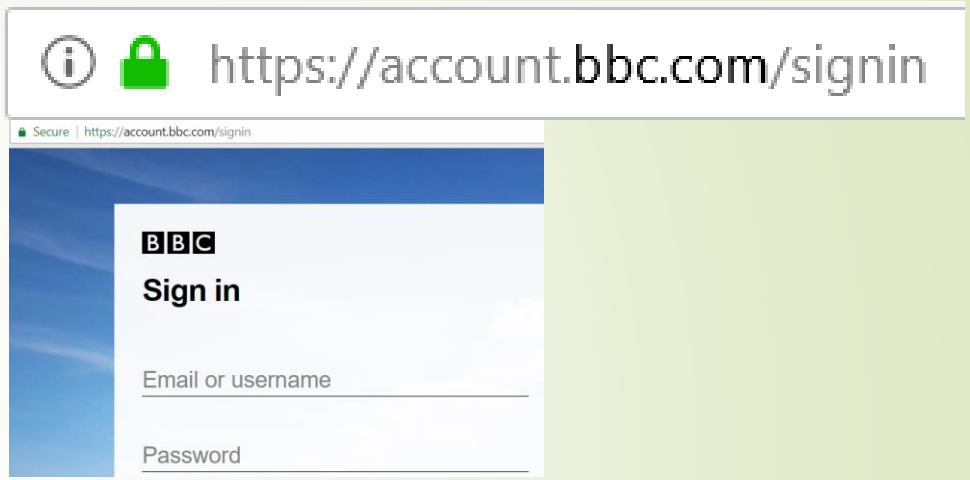
Source: https://en.wikipedia.org/wiki/Public_key_certificate

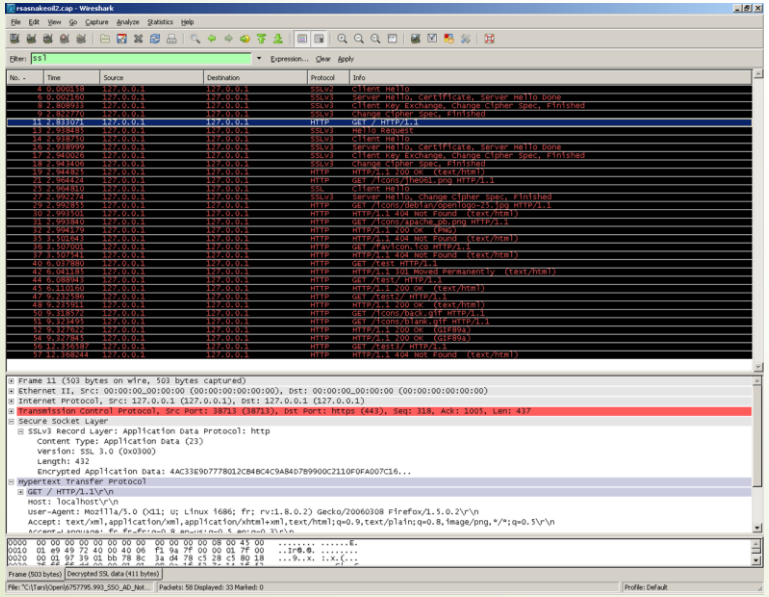
Usage

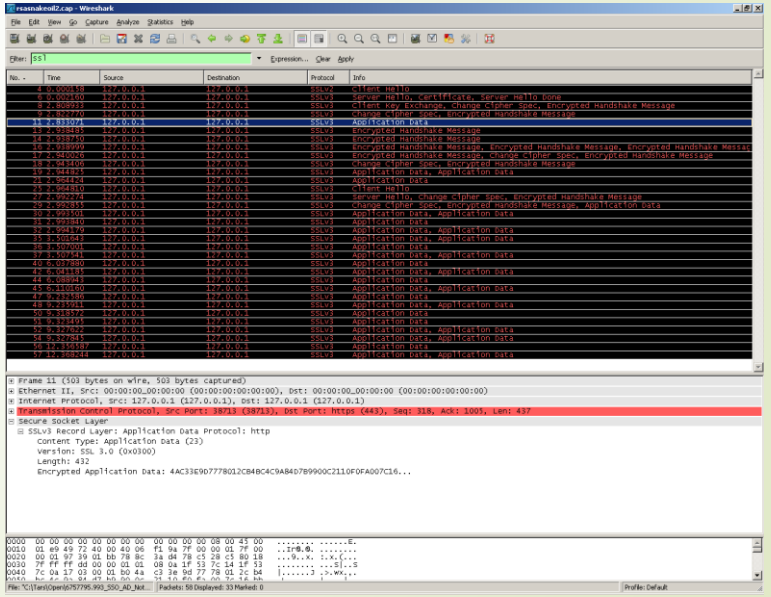
- SSL over HTTP/LDAP
- Identification
- Code Signing
- Encryption/Decryption

SSL over HTTP/LDAP



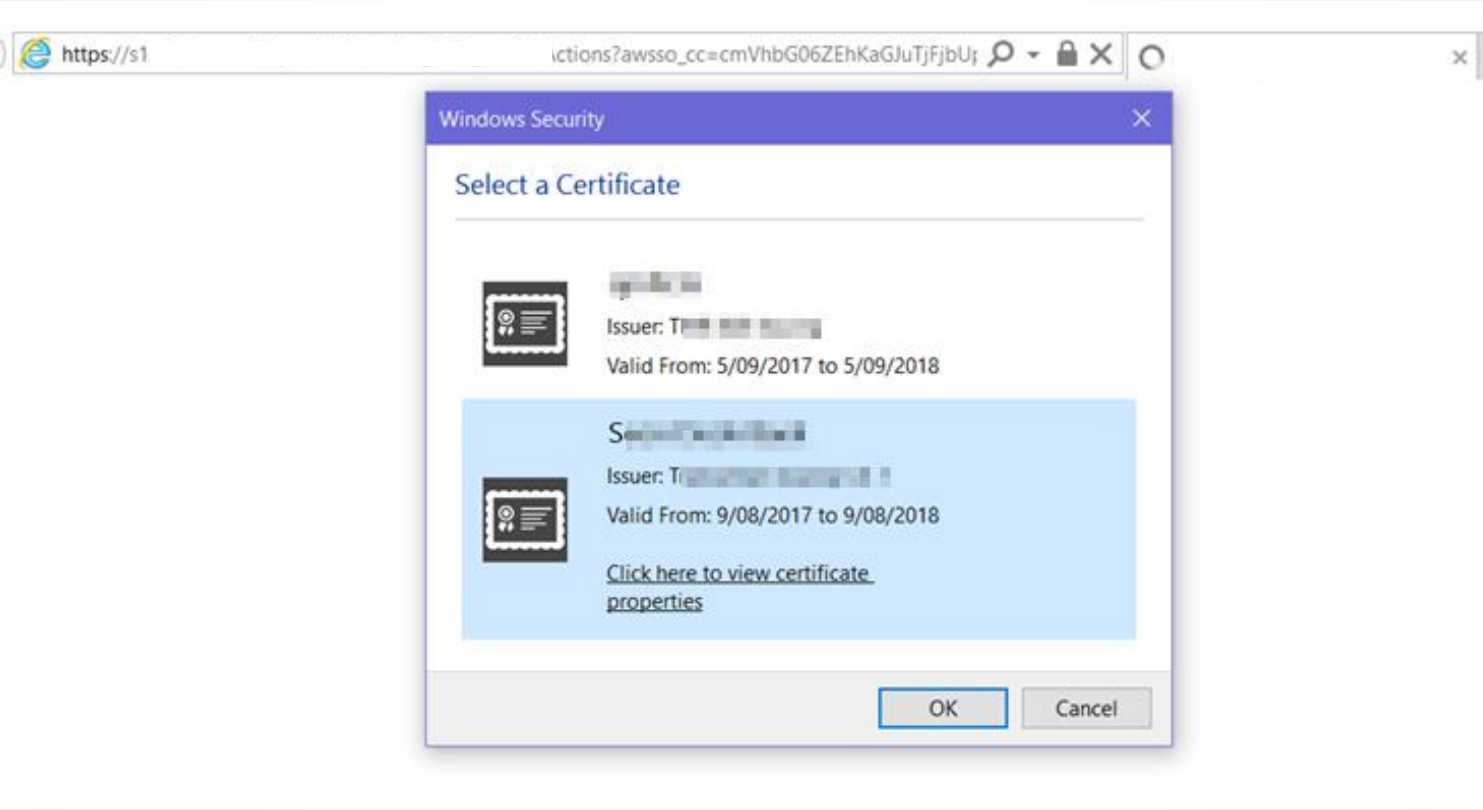






Identification

- Some applications require a certificate for authentication



Code Signing

- Java Web Start applications needs to sign with a valid certificate to assure users they are safe to run

Encryption/Decryption

- A certificate is used to encrypt/decrypt private information in integration with different systems

How To

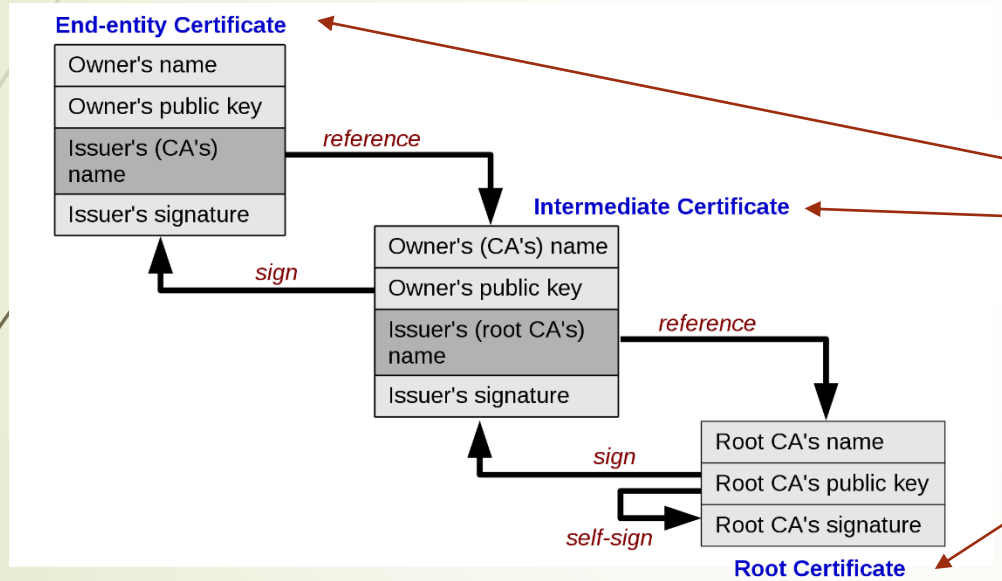
- Check on the fly
- Fill/Fix Broken Certificate Chain
- Extract Certificate on the fly
- Create a New Certificate
- Create s Self-signed Certificate

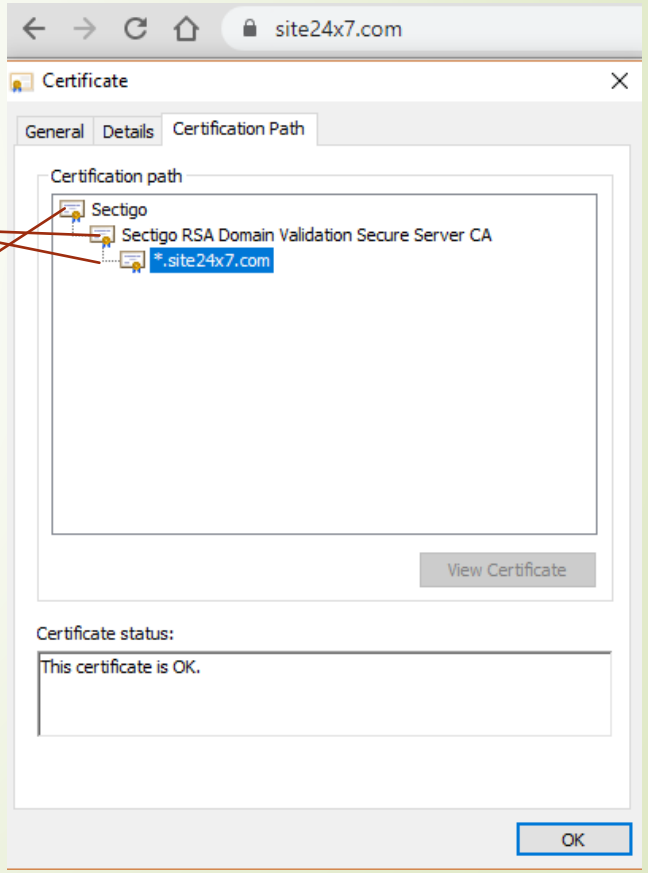
Check on the fly

```
$ curl -verbose https://google.com/ | more
> CONNECT www.google.com:443 HTTP/1.1
...
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/certs/ca-certificates.crt
  CApath: /etc/ssl/certs
...
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
* ALPN, server accepted to use h2
* Server certificate:
*  subject: C=US; ST=California; L=Mountain View; O=Google LLC; CN=www.google.com
*  start date: Feb 12 11:47:41 2020 GMT
*  expire date: May  6 11:47:41 2020 GMT
*  subjectAltName: host "www.google.com" matched cert's "www.google.com"
*  issuer: C=US; O=Google Trust Services; CN=GTS CA 1O1
*  SSL certificate verify ok.
...
```

Fill/Fix Broken Certificate Chain

Complete the chain of trust
by importing (intermediate / root CA certificate) to the **keystore** used by a **software**



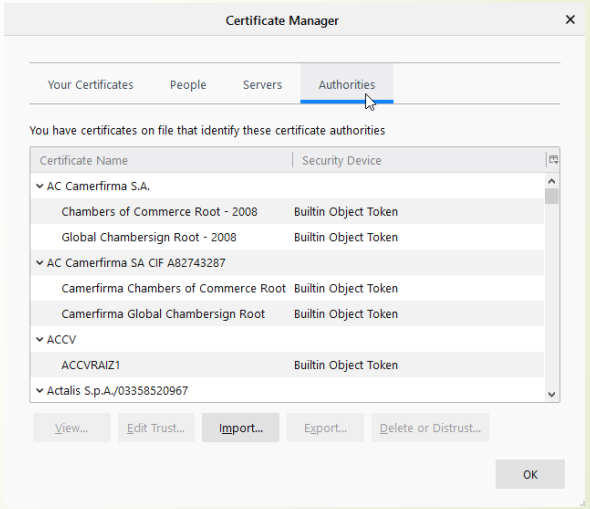


Fill/Fix Broken Certificate Chain

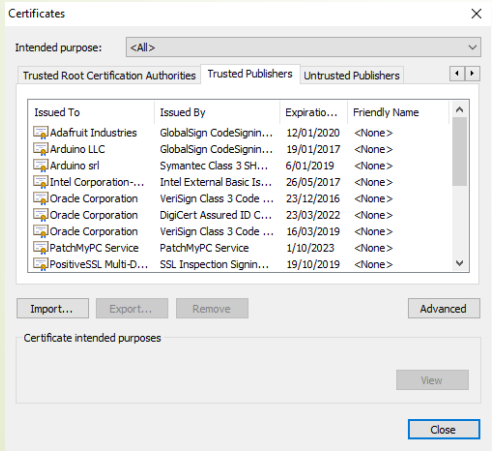
Complete the chain of trust
by **importing** (intermediate / root CA certificate) to the **keystore** used by a **software**

- Web browsers have their own keystore
- Java applications use **.jks (/cacerts)** , which can be specified by environment variable or a config file.

Firefox



Chrome



Eclipse (eclipse.ini)

-Djavax.net.ssl.trustStore=C:\...\...\eclipse\cacerts

Workaround Broken Certificate Chain

Ignore the validation of the certificate chain
by specifying some parameter if available

```
curl -v https://*.com
```

```
* SSL certificate problem: self signed certificate in certificate chain
* stopped the pause stream!
* Closing connection 0
curl: (60) SSL certificate problem: self signed certificate in certificate chain
More details here: https://curl.haxx.se/docs/sslcerts.html
```

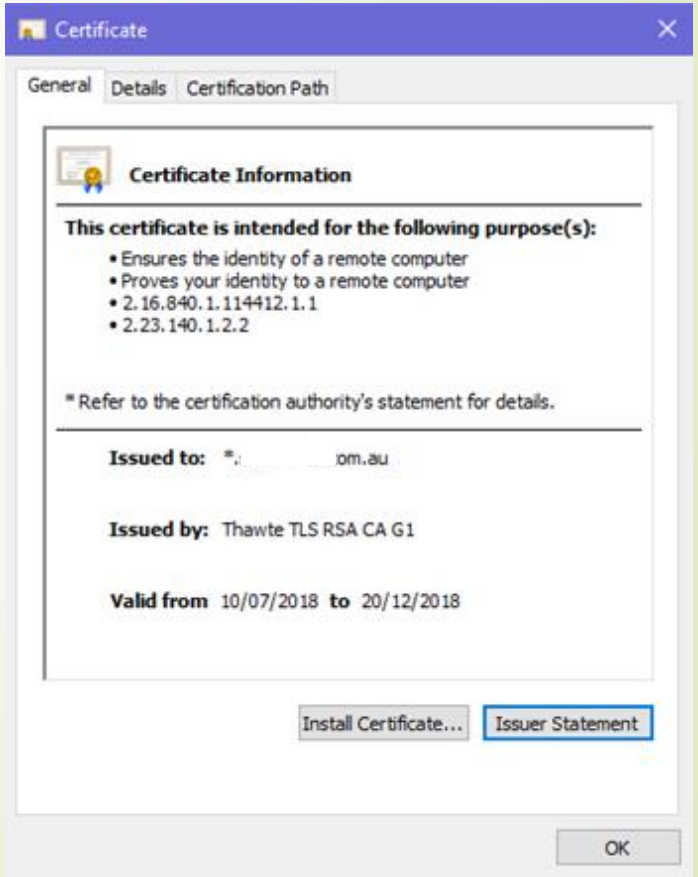
```
curl –v –k https://*.com
```

```
*  SSL certificate verify result: self signed certificate in certificate chain (19), continuing anyway.
> GET / HTTP/1.1
> Host: *.com
> User-Agent: curl/7.58.0
> Accept: */*
```

Extract Certificate on the fly

- echo | openssl s_client -connect hostname:port 2>&1 | sed -ne '/BEGIN CERTIFICATE/,/END CERTIFICATE/p' | openssl x509 -text

```
-----BEGIN CERTIFICATE-----
MIIGOjCCBSKgAwIBAgIQBH/EOFE0/6qXYVH75YfaFjANBgkqhkiG9w0BAQsFADBe
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMR0wGwYDVQQDExRUaGF3dGUgVExTIFJTQSBDQSBHMTAe
Fw0xODA3MTAwMDAwMDBaFw0xODEyMjAxMjAwMDBaMIGbMQswCQYDVQQGEwJBVTEY
MBYGA1UECBMPTmV3IFNvdXRoIFdhbGVzMRMwEQYDVQQHEwpLaW5nc2dyb3ZlMSkw
JwYDVQQKEyBTZW1hIEluZnJhc3RydWN0dXJlIFB0eSBMaW1pdGVkLjEVMBMGA1UE
CxMMSUNUIFNlc
MA0GCSqGSIb3
dcJnK1PEMzTD
hfkPyUCKkP48
ch6SKv3Ha85l
Q/YXV8BCOLUf
fQxwvHwmsBPX
AAGjggK0MIIC
HQ4EFgQULFcv
Y3MuY29tLmF1
HSUEFjAUBggr
cDovL2NkcC50
MEMwNwYJYIZI
ZXJ0LmNvbS9D
AYYYaHR0cDov
Y2FjZXJ0cy50
MAAwggEEBgor
BPkb37jjd80O
3mwWUUS1lHEo
c5UAdgCHdb/n
AwBHMEUCIQDy
GnCygohK9U8N8157Dbs5wefu8nL6aJbStbswDQYJKoZIhvcNAQELBQADggEBACrL
lYSvyQYHvdgmd/x72wMn6jRTsUt2rILLf4RjLh+lcDR4HVGyDMdfjXum7rzHH+UD
QrYHgoqWehky0I9NCjLSrAN+TADkgeD8cbAYiSh3Uy0bzY5jH7skzihnU9+hS/O8
dvqRBlnZNBxBhiqbqaHkzqIK3uEMbS6CO+eK764xsfQ4BEMWvPpkVGodUFWOfFNN
a5Aner3TwgySDYj98jj4IrJpNeTqhy+EVACGH9FRE+Qa5W1Ft3jvJnPbAdVB6iIz
EYHNDFOMtO3ukyoJB7FU2qElG/OdxnilfiKdUCT5dgZxFsDPVLYhalRGZgGYAC6I
X4zo/mNqoNpzXqmpxbc=
-----END CERTIFICATE-----
```



Create a new Certificate

Step 1 : Create Empty Keystore

keytool -genkey -dname <"CN=..., OU=... O=..., L=..., ST=..., C=... "> \
-alias <mykey> -keystore <mykey.jks> -keypass <select a password> \
-storepass <same password as keypass> \
-validity <# of days> -keyalg RSA -keysize 2048

Step 2 : Generate CSR

keytool -certreq -keyalg RSA -alias <mykey> -keystore <mykey.jks> -file <mykey.csr>

Step 3 : Import the signed Certificate into the Keystore created in Step 1

keytool -import -trustcacerts -alias <mykey> -file <mykey.p7b> -keystore <mykey.jks>

Create a Self-signed Certificate

conf file (say box1.conf)

```
[ req ]
default_bits       = 2048
default_keyfile    = server-key.pem
distinguished_name  = subject
req_extensions     = req_ext
x509_extensions    = x509_ext
string_mask        = utf8only

# The Subject DN can be formed using X501 or RFC 4514 (see RFC 4519 for a description).
#   Its sort of a mashup. For example, RFC 4514 does not provide emailAddress.
[ subject ]
countryName        = Country Name (2 letter code)
countryName_default    = US

stateOrProvinceName    = State or Province Name (full name)
stateOrProvinceName_default = NY

localityName           = Locality Name (eg, city)
localityName_default       = New York

organizationName       = Organization Name (eg, company)
organizationName_default   = Example, LLC

# Use a friendly name here because its presented to the user. The server's DNS
#   names are placed in Subject Alternate Names. Plus, DNS names here is deprecated
#   by both IETF and CA/Browser Forums. If you place a DNS name here, then you
#   must include the DNS name in the SAN too (otherwise, Chrome and others that
#   strictly follow the CA/Browser Baseline Requirements will fail).
commonName         = Common Name (e.g. server FQDN or YOUR name)
commonName_default     = Example Company

emailAddress           = Email Address
emailAddress_default       = test@example.com

# Section x509_ext is used when generating a self-signed certificate. I.e., openssl req -x509 ...
[ x509_ext ]

subjectKeyIdentifier       = hash
authorityKeyIdentifier  = keyid,issuer

# You only need digitalSignature below. *If* you don't allow
#   RSA Key transport (i.e., you use ephemeral cipher suites), then
#   omit keyEncipherment because that's key transport.
basicConstraints       = CA:FALSE
keyUsage           = digitalSignature, keyEncipherment
subjectAltName         = @alternate_names
nsComment          = "OpenSSL Generated Certificate"

# RFC 5280, Section 4.2.1.12 makes EKU optional
#   CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
#   In either case, you probably only need serverAuth.
# extendedKeyUsage  = serverAuth, clientAuth

# Section req_ext is used when generating a certificate signing request. I.e., openssl req ...
[ req_ext ]

subjectKeyIdentifier       = hash

basicConstraints       = CA:FALSE
keyUsage           = digitalSignature, keyEncipherment
subjectAltName         = @alternate_names
nsComment          = "OpenSSL Generated Certificate"

# RFC 5280, Section 4.2.1.12 makes EKU optional
#   CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
#   In either case, you probably only need serverAuth.
# extendedKeyUsage  = serverAuth, clientAuth

[ alternate_names ]

DNS.1       = box1.mytoy.mynet
DNS.2       = box2.mytoy.mynet
DNS.3       = box3.mytoy.mynet

# Add these if you need them. But usually you don't want them or
#   need them in production. You may need them for development.
# DNS.5       = localhost
# DNS.6       = localhost.localdomain
# DNS.7       = 127.0.0.1

# IPv6 localhost
# DNS.8     = ::1
```

Create a Self-signed Certificate

- **Generating Self-signed certificate**

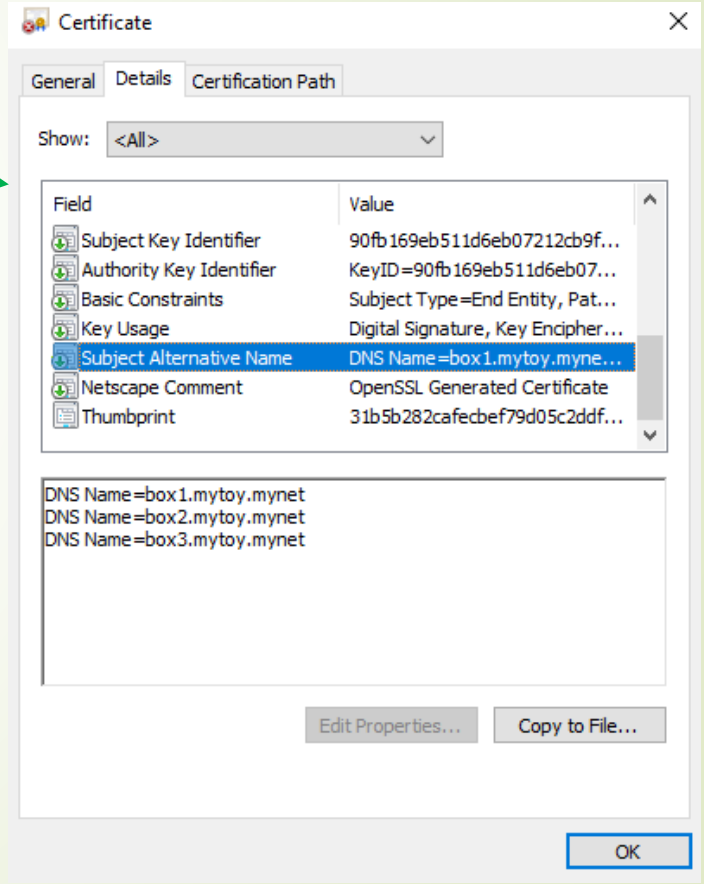
```
openssl req -config box1.conf -new -x509 -sha256 -newkey rsa:2048 -nodes \
-keyout box1.key.pem -out box1.cert.pem -days 3650 \
-subj "/C=AU/ST=Victoria/L=Docklands/O=MYTOY/OU=ORDS/CN=box1.mytoy.mynet"
```

- **Merginng key and cert into PKCS12**

```
openssl pkcs12 -export -in box1.cert.pem \
-inkey box1.key.pem -out box1.p12 \
-name box1 -passin pass:changeit -passout pass:changeit
```

- **Converting PKSC12 into Jave Keystore**

```
keytool -importkeystore -srckeystore box1.p12 -srcstoretype PKCS12 \
-srcstorepass changeit -alias box1 -deststorepass changeit \
-destkeypass changeit -destkeystore box1.jks
```



box1.cert.pem