Public Key Infrastructure (PKI) and x509 Digital Certificates Brief Tutorial

Pratik M Tambe

March 1, 2018

1. What problem does PKI solve?

Consider this scenario:

> Charles: Hey Bob, can you send a confidential email to Alice?
> Bob: Who is that?
> Charles: She's the product manager from ABC Ltd.
> Bob: OK

So Bob tries to find the public key of Alice. He searches the Web and finds several of them. *How can Bob make sure he is using the correct public key from the Alice he wants to contact?*

To solve this problem, we have what we call *certificates*. A certificate is a thing that contains essentially two things:

- A person's name, email addresses, etc.
- A person's public key

So if Bob can find a certificate that says public key A binds to Alice from ABC Ltd., he can make sure that public key A is indeed the correct key to use.

PKI is essentially a whole field that revolves around juggling these things called certificates.

2. What is Certificate Authority?

But wait, what if Bob also finds several certificates that claim to have the public key of Alice? Which one can he trust?

Here comes *Certificate Authorities* (CAs). Basically, there are a dozen of people in the world that we trust to be authoritative. Many OSes come initialized with the information of these CAs. If a CA says a certificate is authentic, we'll believe it.

How does a CA "say" that another certificate is authentic? By *signing* it. Because every system in a Public Key Infrastructure must be seeded with the public keys of trusted CAs, everyone should be able to verify that something is signed by a trusted CA.

So, let us refine our view of what a *certificate* is:

- A person's name, email addresses, etc.
- A person's public key
- The *Issuer* of this certificate.
- The *encrypted hash* of this certificate. (So that a receiver can verify the signature of the issuer.)

Let's look at an example scenario:

1. Bob receives certificate X; he looks at it, it claims to be Alice's certificate and is issued by A.
2. In order for Bob to verify that X is valid, he downloads the public key of A from somewhere, and decrypts the *encrypted hash* inside X to see if it matches. If it matches, then X is valid.
3. Now the question is, Bob just downloaded A's certificate from somewhere, how does he know it's valid? A's certificate is in turned signed by B, which is a trusted CA that is seeded from his computer installation. So he just uses B's public key to verify that the version of A's certificate that he used, was indeed issued by B.
4. Since B issued A issued X, we can say B trusts A trusts X. Since B is a trusted CA, we'll trust what he trusts. So Bob can safely trusts the authenticity of X.

The exact method that Bob can use to obtain the certificate for A is not specified by RFC 5280. What that basically means is that people are free to use whatever protocols that makes sense. For example, in PDF documents, it's common practice to dump the whole chain of certificates required to verify the signer, so a viewer does not need to go downloading every certificate in the *chain*.

3. Who issues Certificates to CAs?

It's the chicken and egg problem. It turns out that CAs can sign their own certificates. These are called *self-signed certificates*. Everyone is supposed to obtain their public keys through secure means (going to CAs' offices and meeting them personally, for example), because PKI alone cannot ensure the validity of any self-signed certificates. We usually refer to the "top level" CAs as Root CAs. And we usually refer to this type of "going to their offices" type of thing *out-of-band* (e.g. out-of-band distribution of certificates means distributed non-electronically).

Most of the time, though, this is less rigorous than that. For example, your Web browser usually comes seeded with about 30+ Root CAs. The browsers' vendors have nominated who are trustworthy and so by using the browser, you trust their decisions.

4. What is the difference between public key algorithm and (digital) signature algorithm?

Encryption is a mechanism by which a message is transformed so that only the sender and recipient can see. For instance, suppose that Alice wants to send a private message to Bob. To do so, she first needs Bob's public-key; since everybody can see his public-key, Bob can send it over the network in the clear without any concerns. Once Alice has Bob's public-key, she

encrypts the message using Bob's public-key and sends it to Bob. Bob receives Alice's message and, using his private-key, decrypts it. Public key algorithm is used here.
Digital signature is a mechanism by which a message is authenticated i.e. proving that a message is effectively coming from a given sender, much like a signature on a paper document. For instance, suppose that Alice wants to digitally sign a message to Bob. To do so, she uses her private-key to encrypt the message; she then sends the message along with her public-key (typically, the public key is attached to the signed message). Since Alice's public-key is the only key that can decrypt that message, a successful decryption constitutes a Digital Signature Verification, meaning that there is no doubt that it is Alice's private key that encrypted the message.
The two previous paragraphs illustrate the encryption/decryption and signature/verification principles. Both encryption and digital signature can be combined, hence providing privacy and authentication. As mentioned earlier, symmetric-key plays a major role in public-key encryption implementations. This is because asymmetric-key encryption algorithms are somewhat slower than symmetric-key algorithms
For Digital signature, another technique used is called hashing. Hashing produces a message digest that is a small and unique representation (a bit like a sophisticated checksum) of the complete message. Hashing algorithms are a one-way encryption, i.e. it is impossible to derive the message from the digest. The main reasons for producing a message digest are:
1 The message integrity being sent is preserved; any message alteration will immediately be detected;
2 The digital signature will be applied to the digest, which is usually considerably smaller than the message itself;
3 Hashing algorithms are much faster than any encryption algorithm (asymmetric or symmetric).

I cannot explain better than the following guide**.**

http://www.cgi.com/files/white-papers/cgi_whpr_35_pki_e.pdf

Step 1: Generate a Private Key.

The first step is to create your RSA Private Key. This key is a 1024 bit RSA key (1024 bit modulus) which is encrypted using Triple-DES and stored in a PEM format so that it is readable as ASCII text.

```
openssl genrsa -des3 -out toyCA.key 1024
```

Now enter the above password in toyCA.pass. We use 'Pas55w0rd' as the password value for all passwords for this primer.

```
echo "Pa55w0rd" > toyCA.pass
```

Note: One could use 'gendsa' above too
Supported algorithms that I am aware of: [-aes128] [-aes192] [-aes256] [-camellia128] [-camellia192] [-camellia256] [-des] [-des3]

Make a copy in priv extension

```
cp toyCA.key toyCA.priv
```

Step 2: Generate a CSR (Certificate Signing Request)

```
openssl req -new -key toyCA.key -out toy.csr
```

Then Enter the information

Country Name (2 letter code) [XX]:US
State or Province Name (full name) []:Maryland
Locality Name (eg, city) [Default City]:Silver Spring
Organization Name (eg, company) [Default Company Ltd]:Company
Organizational Unit Name (eg, section) []:Some Work
Common Name (eg, your name or your server's hostname) []:company.com
Email Address []:toy@company.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: Pa55w0rd
An optional company name []: Pa55w0rd

Step 3: Remove Passphrase from Key (Optional but not recommended) - Caution: This will remove the Triple-DES encryption from the key
#Good idea to make backup

```
cp toyCA.key toyCA.key.org
openssl rsa -in toyCA.key.org -out toyCA.key
```

Step 4: Generating a Self-Signed Certificate (in either CRT or PEM format)

```
openssl x509 -req -days 365 -in toy.csr -signkey toyCA.key -out toyCA.crt
openssl x509  -in  toyCA.crt -out toyCA.pem
```

Note: The -signkey will be commercial CA root/intermediate key e.g. Thawte, Verisign in most professional certs

To view the content of cert and private key

```
openssl x509 -in toyCA.pem  -text -noout
openssl x509 -in toyCA.pem -subject -issuer -nameopt multiline,show_type -
noout -subject_hash -issuer_hash
openssl rsa -in toyCA.priv -text -noout
```

To verify that the private key and public key belong to a cert chain

```
openssl rsa -noout -modulus -in toyCA.priv -passin file:toyCA.pass | openssl
md5
openssl x509 -noout -modulus -in toyCA.pem | openssl md5
```

should have same md5 checksum.

Note: To select dsa algorithm with SHA1 we use -dsaWithSHA1 option (see the complete list in FAQs below)

```
openssl x509 -req -dsaWithSHA1  -days 365 -in toy.csr -signkey toyCA.key -out
toyCA_dsa.crt
```

Step 5: Prepare to create an Intermediate Cert

First generate the private key for intermediate Cert (This is the Cert one may choose to sign the chain of trust of certificates. In the even the entire chain of trust of certificates is compromised our root CA is still protected in a safe place since Intermediate CA can be discarded)

```
openssl genrsa -des3 -out toyIntermediate.key 1024
```

(Optional) Then generate decrypted/ unencrypted private key - Caution: This will remove the Triple-DES encryption from the key

```
openssl rsa -in toyIntermediate.key -out toyDecryptedIntermediate.key
openssl req -new -key toyDecryptedIntermediate.key -out
toyDecryptedIntermediate.csr
```

Then Enter the information

Country Name (2 letter code) [XX]:IN
State or Province Name (full name) []:Maharashtra
Locality Name (eg, city) [Default City]:Mumbai
Organization Name (eg, company) [Default Company Ltd]:Company India
Organizational Unit Name (eg, section) []:Some Work
Common Name (eg, your name or your server's hostname) []:company.com
Email Address []:toy@company.in

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: Pa55w0rd
An optional company name []: Pa55w0rd

Step 6: Generate an intermediate Certificate

```
openssl x509 -req -days 3650 -in toyDecryptedIntermediate.csr -CA toyCA.pem -
CAkey toyCA.key -set_serial 01 -out toyIntermediate.crt
openssl x509  -in  toyIntermediate.crt -out toyIntermediate.pem
```

Notice: How subject and issuer contents are different

```
openssl x509 -in toyIntermediate.pem  -text -noout
```

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 1 (0x1)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, ST=Maryland, L=Silver Spring, O=Company, OU=Some Work, CN=company.com/emailAddress=toy@company.com
        Validity
            Not Before: Jun  2 15:56:45 2016 GMT
            Not After : May 31 15:56:45 2026 GMT
        Subject: C=IN, ST=Maharashtra, L=Mumbai, O=Company India, OU=Some Work, CN=company.com/emailAddress=toy@company.in
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    00:ea:39:3b:a8:4a:9f:d4:f5:17:8c:d1:93:70:e3:
                    e0:e7:cf:6e:85:75:23:b7:e1:36:af:9f:05:1d:69:
                    b4:60:cb:01:29:be:c2:c6:de:8c:c2:03:10:24:70:
                    3b:c3:5b:9b:4a:25:ab:db:af:55:fc:90:62:dd:bc:
                    3d:f4:a7:40:1d:cc:48:74:a4:68:d9:bd:61:9f:0d:
                    c3:97:d7:ec:1b:7f:a6:ae:5d:8a:3d:86:5b:89:52:
                    e5:89:2d:f0:ac:e8:03:dd:f9:37:4c:2e:60:df:d5:
                    6e:92:06:a6:b3:00:3f:35:74:5a:93:0f:a1:5f:59:
                    cc:8d:49:c2:a0:6e:d1:24:3b
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
         24:e6:e7:0c:2c:93:8c:d2:c9:fe:21:47:b7:3f:43:0d:12:89:
         9d:42:0d:ac:ac:a3:89:60:33:3e:01:a3:7c:7a:c8:e2:28:15:
         bb:26:10:b8:33:79:82:2a:12:25:f6:a4:4e:0c:a6:e2:2a:f7:
         d6:c0:25:73:c2:bb:5e:67:ff:cd:01:10:ff:e1:52:7b:8d:d0:
         e2:96:5a:c9:17:51:3f:cf:60:3c:af:3b:66:85:04:08:97:94:
         d6:86:d1:ec:f7:20:db:d7:c9:4a:d5:33:90:a8:e0:7c:63:02:
         f5:8f:d9:e0:15:29:4e:97:5a:a2:e0:12:fd:ea:69:d0:fc:3e:
         42:cb
```

Step 7: Repeat Steps 5 and step 6 to create chain of trust but this time with intermediate CA parameters instead of root CA parameters. This means one has to generate a private key first. Then a CSR. Then use the private key of the Intermediate CA to sign this.

Step 8: Revoke a certificate using a Certificate Revocation List (CRL) in case of a compromise of a set of certificates.
Step (A) Prepare to create a CRL (Give Database)

```
touch certindex
echo 01 > certserial
echo 01 > crlnumber
```

Then copy and paste the following in toyCA.conf

```
# Mainly copied from:
# http://swearingscience.com/2009/01/18/openssl-self-signed-ca/

[ ca ]
default_ca = myca

[ crl_ext ]
# issuerAltName=issuer:copy  #this would copy the issuer name to altname
authorityKeyIdentifier=keyid:always

 [ myca ]
 dir = ./
 new_certs_dir = $dir
 unique_subject = no
 certificate = $dir/toyCA.pem
 database = $dir/certindex
 private_key = $dir/toyCA.priv
 serial = $dir/certserial
 default_days = 730
 default_md = sha1
 policy = myca_policy
 x509_extensions = myca_extensions
 crlnumber = $dir/crlnumber
 default_crl_days = 730

 [ myca_policy ]
 commonName = supplied
 stateOrProvinceName = supplied
 countryName = optional
 emailAddress = optional
 organizationName = supplied
 organizationalUnitName = optional

 [ myca_extensions ]
 basicConstraints = CA:false
 subjectKeyIdentifier = hash
 #authorityKeyIdentifier = keyid:always
 keyUsage = digitalSignature,keyEncipherment
 extendedKeyUsage = serverAuth
 crlDistributionPoints = URI:http://company.com/root.crl
 subjectAltName  = @alt_names

 [alt_names]
 DNS.1 = company.com
 DNS.2 = *.company.com
```

Then sign the request

```
openssl ca -batch -config toyCA.conf -notext -in toy.csr -out toyIA.crt
```

This will output
Using configuration from toyCA.conf
Enter pass phrase for .//toyCA.priv:

```
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName           :PRINTABLE:'US'
stateOrProvinceName   :ASN.1 12:'Maryland'
localityName          :ASN.1 12:'Silver Spring'
organizationName      :ASN.1 12:'Company'
organizationalUnitName:ASN.1 12:'Some Work'
commonName            :ASN.1 12:'company.com'
emailAddress          :IA5STRING:'toy@company.com'
Certificate is to be certified until Jun  2 16:34:07 2018 GMT (730 days)
```

Step (B) Create a CRL

```
openssl ca -config toyCA.conf -gencrl -keyfile toyCA.key -cert toyCA.crt -out
root.crl.pem
openssl crl -inform PEM -in root.crl.pem -outform DER -out root.crl
rm root.crl.pem
```

Now copy this root.crl file to http://company.com/root.crl

Step (C) To revoke a certificate

```
openssl ca -config toyCA.conf -revoke <certificate to revoke> -keyfile
toyCA.key -cert toyCA.crt
```

FAQs

1) Can I do even more efficient cryptography?

Yes!

Using **Elliptic Curve Cryptography (ECC)**

```
openssl ecparam -out key.pem -name prime256v1 -genkey
```

See http://www.secg.org/ for more information

openssl can provide full list of EC parameter names suitable for
passing to the -name option above:

openssl ecparam -out key.pem -name prime256v1 -genkey

```
openssl ecparam -list_curves
```

```
secp112r1 : SECG/WTLS curve over a 112 bit prime field
secp112r2 : SECG curve over a 112 bit prime field
secp128r1 : SECG curve over a 128 bit prime field
secp128r2 : SECG curve over a 128 bit prime field
secp160k1 : SECG curve over a 160 bit prime field
secp160r1 : SECG curve over a 160 bit prime field
secp160r2 : SECG/WTLS curve over a 160 bit prime field
secp192k1 : SECG curve over a 192 bit prime field
secp224k1 : SECG curve over a 224 bit prime field
secp224r1 : NIST/SECG curve over a 224 bit prime field
secp256k1 : SECG curve over a 256 bit prime field
secp384r1 : NIST/SECG curve over a 384 bit prime field
secp521r1 : NIST/SECG curve over a 521 bit prime field
prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field
prime192v2: X9.62 curve over a 192 bit prime field
prime192v3: X9.62 curve over a 192 bit prime field
prime239v1: X9.62 curve over a 239 bit prime field
prime239v2: X9.62 curve over a 239 bit prime field
prime239v3: X9.62 curve over a 239 bit prime field
prime256v1: X9.62/SECG curve over a 256 bit prime field
sect113r1 : SECG curve over a 113 bit binary field
sect113r2 : SECG curve over a 113 bit binary field
sect131r1 : SECG/WTLS curve over a 131 bit binary field
sect131r2 : SECG curve over a 131 bit binary field
sect163k1 : NIST/SECG/WTLS curve over a 163 bit binary field
```

sect163r1 : SECG curve over a 163 bit binary field
sect163r2 : NIST/SECG curve over a 163 bit binary field
sect193r1 : SECG curve over a 193 bit binary field
sect193r2 : SECG curve over a 193 bit binary field
sect233k1 : NIST/SECG/WTLS curve over a 233 bit binary field
sect233r1 : NIST/SECG/WTLS curve over a 233 bit binary field
sect239k1 : SECG curve over a 239 bit binary field
sect283k1 : NIST/SECG curve over a 283 bit binary field
sect283r1 : NIST/SECG curve over a 283 bit binary field
sect409k1 : NIST/SECG curve over a 409 bit binary field
sect409r1 : NIST/SECG curve over a 409 bit binary field
sect571k1 : NIST/SECG curve over a 571 bit binary field
sect571r1 : NIST/SECG curve over a 571 bit binary field
c2pnb163v1: X9.62 curve over a 163 bit binary field
c2pnb163v2: X9.62 curve over a 163 bit binary field
c2pnb163v3: X9.62 curve over a 163 bit binary field
c2pnb176v1: X9.62 curve over a 176 bit binary field
c2tnb191v1: X9.62 curve over a 191 bit binary field
c2tnb191v2: X9.62 curve over a 191 bit binary field
c2tnb191v3: X9.62 curve over a 191 bit binary field
c2pnb208w1: X9.62 curve over a 208 bit binary field
c2tnb239v1: X9.62 curve over a 239 bit binary field
c2tnb239v2: X9.62 curve over a 239 bit binary field
c2tnb239v3: X9.62 curve over a 239 bit binary field
c2pnb272w1: X9.62 curve over a 272 bit binary field
c2pnb304w1: X9.62 curve over a 304 bit binary field
c2tnb359v1: X9.62 curve over a 359 bit binary field
c2pnb368w1: X9.62 curve over a 368 bit binary field
c2tnb431r1: X9.62 curve over a 431 bit binary field
wap-wsg-idm-ecid-wtls1: WTLS curve over a 113 bit binary field
wap-wsg-idm-ecid-wtls3: NIST/SECG/WTLS curve over a 163 bit binary field
wap-wsg-idm-ecid-wtls4: SECG curve over a 113 bit binary field
wap-wsg-idm-ecid-wtls5: X9.62 curve over a 163 bit binary field
wap-wsg-idm-ecid-wtls6: SECG/WTLS curve over a 112 bit prime field
wap-wsg-idm-ecid-wtls7: SECG/WTLS curve over a 160 bit prime field
wap-wsg-idm-ecid-wtls8: WTLS curve over a 112 bit prime field
wap-wsg-idm-ecid-wtls9: WTLS curve over a 160 bit prime field
wap-wsg-idm-ecid-wtls10: NIST/SECG/WTLS curve over a 233 bit binary field
wap-wsg-idm-ecid-wtls11: NIST/SECG/WTLS curve over a 233 bit binary field
wap-wsg-idm-ecid-wtls12: WTLS curvs over a 224 bit prime field
Oakley-EC2N-3:
  IPSec/IKE/Oakley curve #3 over a 155 bit binary field.
  Not suitable for ECDSA.
  Questionable extension field!
Oakley-EC2N-4:
  IPSec/IKE/Oakley curve #4 over a 185 bit binary field.

Not suitable for ECDSA.
Questionable extension field!

2) Which ciphers do openssl support?

```
openssl ciphers
```

ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-DSS-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-SHA256:DHE-RSA-AES256-SHA:DHE-DSS-AES256-SHA:DHE-RSA-CAMELLIA256-SHA:DHE-DSS-CAMELLIA256-SHA:ECDH-RSA-AES256-GCM-SHA384:ECDH-ECDSA-AES256-GCM-SHA384:ECDH-RSA-AES256-SHA384:ECDH-ECDSA-AES256-SHA384:ECDH-RSA-AES256-SHA:ECDH-ECDSA-AES256-SHA:AES256-GCM-SHA384:AES256-SHA256:AES256-SHA:CAMELLIA256-SHA:PSK-AES256-CBC-SHA:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:DHE-DSS-AES128-GCM-SHA256:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES128-SHA256:DHE-DSS-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA:ECDHE-RSA-DES-CBC3-SHA:ECDHE-ECDSA-DES-CBC3-SHA:DHE-RSA-SEED-SHA:DHE-DSS-SEED-SHA:DHE-RSA-CAMELLIA128-SHA:DHE-DSS-CAMELLIA128-SHA:EDH-RSA-DES-CBC3-SHA:EDH-DSS-DES-CBC3-SHA:ECDH-RSA-AES128-GCM-SHA256:ECDH-ECDSA-AES128-GCM-SHA256:ECDH-RSA-AES128-SHA256:ECDH-ECDSA-AES128-SHA256:ECDH-RSA-AES128-SHA:ECDH-ECDSA-AES128-SHA:ECDH-RSA-DES-CBC3-SHA:ECDH-ECDSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES128-SHA256:AES128-SHA:SEED-SHA:CAMELLIA128-SHA:DES-CBC3-SHA:IDEA-CBC-SHA:PSK-AES128-CBC-SHA:PSK-3DES-EDE-CBC-SHA:KRB5-IDEA-CBC-SHA:KRB5-DES-CBC3-SHA:KRB5-IDEA-CBC-MD5:KRB5-DES-CBC3-MD5:ECDHE-RSA-RC4-SHA:ECDHE-ECDSA-RC4-SHA:ECDH-RSA-RC4-SHA:ECDH-ECDSA-RC4-SHA:RC4-SHA:RC4-MD5:PSK-RC4-SHA:KRB5-RC4-SHA:KRB5-RC4-MD5

and related commands

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

aes-128-cbc
aes-128-ecb
aes-192-cbc
aes-192-ecb
aes-256-cbc
aes-256-ecb
base64
bf
bf-cbc

bf-cfb
bf-ecb
bf-ofb
camellia-128-cbc
camellia-128-ecb
camellia-192-cbc
camellia-192-ecb
camellia-256-cbc
camellia-256-ecb
cast
cast-cbc
cast5-cbc
cast5-cfb
cast5-ecb
cast5-ofb
des
des-cbc
des-cfb
des-ecb
des-ede
des-ede-cbc
des-ede-cfb
des-ede-ofb
des-ede3
des-ede3-cbc
des-ede3-cfb
des-ede3-ofb
des-ofb
des3
desx
idea
idea-cbc
idea-cfb
idea-ecb
idea-ofb
rc2
rc2-40-cbc
rc2-64-cbc
rc2-cbc
rc2-cfb
rc2-ecb
rc2-ofb
rc4
rc4-40
seed
seed-cbc

seed-cfb
seed-ecb
seed-ofb
zlib

3) Can I see all Openssl public key algorithms? Note: This is really helpful when generating certificate.

```
openssl list-public-key-algorithms
```

Name: OpenSSL RSA method
    Type: Builtin Algorithm
    OID: rsaEncryption
    PEM string: RSA
Name: rsa
    Type: Alias to rsaEncryption
Name: OpenSSL PKCS#3 DH method
    Type: Builtin Algorithm
    OID: dhKeyAgreement
    PEM string: DH
Name: dsaWithSHA
    Type: Alias to dsaEncryption
Name: dsaEncryption-old
    Type: Alias to dsaEncryption
Name: dsaWithSHA1-old
    Type: Alias to dsaEncryption
Name: dsaWithSHA1
    Type: Alias to dsaEncryption
Name: OpenSSL DSA method
    Type: Builtin Algorithm
    OID: dsaEncryption
    PEM string: DSA
Name: OpenSSL EC algorithm
    Type: Builtin Algorithm
    OID: id-ecPublicKey
    PEM string: EC
Name: OpenSSL HMAC method
    Type: Builtin Algorithm
    OID: hmac
    PEM string: HMAC
Name: OpenSSL CMAC method
    Type: Builtin Algorithm
    OID: cmac
    PEM string: CMAC

4) Can I see all Openssl message digest algorithms?

```
Openssl list-message-digest-algorithms
```

DSA
DSA-SHA
DSA-SHA1 => DSA
DSA-SHA1-old => DSA-SHA1
DSS1 => DSA-SHA1
MD4
MD5
RIPEMD160
RSA-MD4 => MD4
RSA-MD5 => MD5
RSA-RIPEMD160 => RIPEMD160
RSA-SHA => SHA
RSA-SHA1 => SHA1
RSA-SHA1-2 => RSA-SHA1
RSA-SHA224 => SHA224
RSA-SHA256 => SHA256
RSA-SHA384 => SHA384
RSA-SHA512 => SHA512
SHA
SHA1
SHA224
SHA256
SHA384
SHA512
DSA
DSA-SHA
dsaWithSHA1 => DSA
dss1 => DSA-SHA1
ecdsa-with-SHA1
MD4
md4WithRSAEncryption => MD4
MD5
md5WithRSAEncryption => MD5
ripemd => RIPEMD160
RIPEMD160
ripemd160WithRSA => RIPEMD160
rmd160 => RIPEMD160
SHA
SHA1
sha1WithRSAEncryption => SHA1

SHA224
sha224WithRSAEncryption => SHA224
SHA256
sha256WithRSAEncryption => SHA256
SHA384
sha384WithRSAEncryption => SHA384
SHA512
sha512WithRSAEncryption => SHA512
shaWithRSAEncryption => SHA
ssl2-md5 => MD5
ssl3-md5 => MD5
ssl3-sha1 => SHA1
whirlpool

5) Can you explain various PKI technical Jargons briefly?

X.509 This is the protocol that specifies most of these things.

ASN.1 It's the syntax used to describe the things in a certificate. If certificates were written in XML, then ASN.1 would be the schema's syntax. (This is over-simplified. ASN.1 can indeed use XML. Wikipedia's page on ASN.1 actually sums it up quite well.).

PKIX An organization that writes RFCs on these things.

Algorithm When used in a PKI context, this means things like "RSA with SHA1", "DSA with SHA1", etc. If you've read up on cryptographic signing, you'll know that we need to 1) hash something 2) encrypt the hash. "RSA with SHA1" would mean that we hash with SHA1, and then encrypt with RSA.

Object Identifiers (OIDs)OK this is messy. They figured out that we don't want to use English to describe something in a X.509 certificate. So they came up with numbers. For example, when we want to refer to "RSA" in a certificate, we don't put in the string "RSA". Instead, we'll use its OID "1.2.840.11359". You can read about registered OIDs here.

DER Distinguished Encoding Rule. It's the format that a certificate is used to implement what's promised in the ASN.1 specification. It's easy to understand when you know there is another encoding rule called XER -- XML Encoding Rule.

PEM Privacy Enhanced Mail. This was supposed to be another format (encoding rule) to encode a certificate, in clear text. Nowadays, though, when people say PEM it usually means DER further encoded to Base64 (using only bytes in the range of displayable characters, so it is suitable for distribution through emails.)

DN Distinguished Name. A unique name to identify someone. For example, Karen is probably not really useful to identify someone, so we'll say something like CN=Karen Berge,CN=admin,DC=corp,DC=Fabrikam,DC=COM which looks self explanatory.

SPKI Simple PKI. The PKI we just described (the RFC 5280 family) binds a certificate with a distinguished name. SPKI describes certificates that bind a public key to a set of permissions. Not many people actually use this as far as I know.

PKCS Public Key Cryptography Standard. You usually see people say PKCS #7 or PKCS #12. These are different chapters of the same standard. For example, PKCS #7 describes how digital signatures should work; PKCS # 12 describes the format that stores a certificate and private key together, etc. I've listed some common and important PKCS standards below.

PKCS #1 An RSA public key usually contains two numbers; a private key usually contains one number (the key). This is the file format to describe how to store those numbers in a file.

PKCS #3 Describes the Diffie-Hellman Key Exchange mechanism.

PKCS #7/CMS Cryptographic Message Syntax. Describes the actual message that gets signed and/or encrypted. Think of this as the specification of a TCP packet -- some header information wraps the actual data.

PKCS #8 Used to carry private certificate keypairs (encrypted or unencrypted). This standard describes syntax for private-key information, including a private key for some public-key algorithm and a set of attributes. The standard also describes syntax for encrypted private keys. The intention of including a set of attributes is to provide a simple way for a user to establish trust in information such as a distinguished name or a top-level certification authority's public key.

PKCS #9 A standard on the "meta-data" on a certificate. For example, a certificate can specify that it is only valid for "signing", etc. There is an extension field in PKCS #7 to specify those purposes, and PKCS #9 is the standard format of how to specify those.

PKCS #10 When someone sends his certificate to be signed by a Root CA, he's said to be sending a certificate signing request. There are some meta-data that needs to be wrapped and is specified by this PKCS.

PKCS #12 The file format that stores a certificate and a private key together in one file.

CLR Certificate Revocation List. These lists are hosted by some central authorities' servers to say which certificates have been revoked, for whatever reason. This works like credit cards' revocation. For example, someone might have his private key stolen and want the certificate revoked. If you think "that would put huge loads on those central servers", you're right. People are still trying to come up with better strategies.OCSPOnline Certificate Status Protocol. An attempt to improve the CLR approach to find out a certificate's revocation status.CSRCertificate Signing Request. See PKCS #10 above.

For more information see.

http://www.emclink.net/emc-plus/rsa-labs/standards-initiatives/public-key-cryptography-standards.htm

6) How does SSL handshake work? See diagram/ illustration below.

1. The client sends a Hello message to the server.

   The message includes a list of algorithms supported by the client and a random number that will be used to generate the keys.

2. The server responds by sending a Hello message to the client. This message includes:
   - The algorithm to use. The server selected this from the list sent by the client.
   - A random number, which will be used to generate the keys.
3. The server sends its certificate to the client.
4. The client authenticates the server by checking the validity of the server's certificate, the issuer CA, and optionally, by checking that the host name of the server matches the subject DN. The client sends a Session ID for session caching.
5. The client generates a random value ("pre-master secret"), encrypts it using the server's public key, and sends it to the server.
6. The server uses its private key to decrypt the message to retrieve the pre-master secret.
7. The client and server separately calculate the keys that will be used in the SSL session.

   These keys are not sent to each other because the keys are calculated based on the pre-master secret and the random numbers, which are known to each side. The keys include:

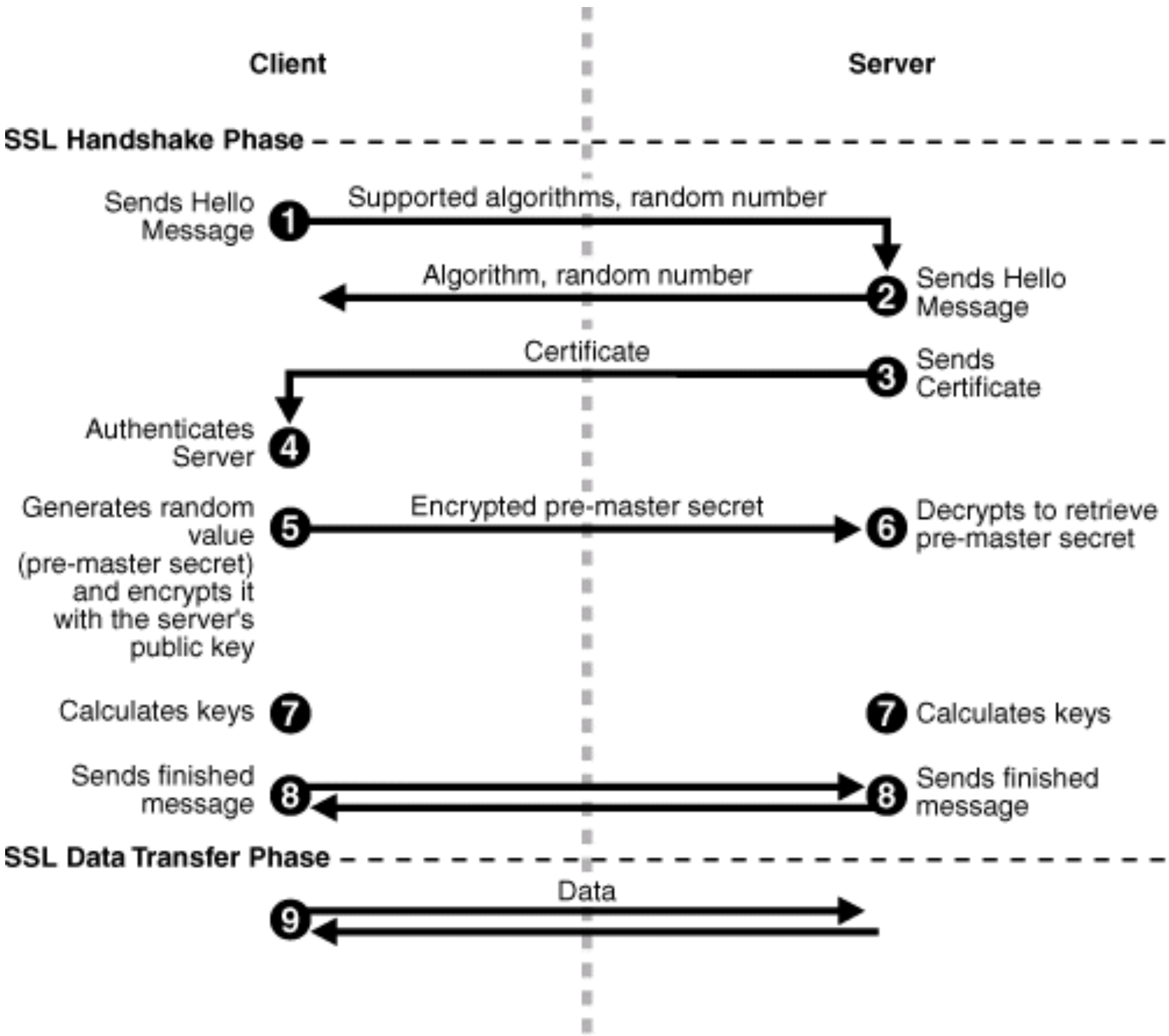
   - Encryption key that the client uses to encrypt data before sending it to the server
   - Encryption key that the server uses to encrypt data before sending it to the client
   - Key that the client uses to create a message digest of the data
   - Key that the server uses to create a message digest of the data

   The encryption keys are symmetric, that is, the same key is used to encrypt and decrypt the data.

8. The client and server send a `Finished` message to each other. These are the first messages that are sent using the keys generated in the previous step (the first "secure" messages).

   The `Finished` message includes all the previous handshake messages that each side sent. Each side verifies that the previous messages that it received match the messages included in the `Finished` message. This checks that the handshake messages were not tampered with.

The session key (step 7 in diagram) is symmetric key e.g AES-256. So actual messages are encrypted and decrypted by symmetric key. Certificate (step 3 - server public key) in our case use RSA as public key algorithm and same certificate DSA with SHA1(cryptographic hash function) as digital signature algorithm. Server never exposes its private key outside itself. Nor does client expose private key outside itself. private keys are used at the respective nodes (client and server) only for validation of cert from the other node. The premaster secret encrypts using the private key algorithm - In our case RSA.




```
# example if this was certificate called SO.pem containing only public key
and test_SO.priv containing private key connecting to some Load balancer

openssl s_client -key test_SO.priv -cert SO.pem -showcerts -connect
c2.company.net:443 -state -debug
```

```
# example if this was certificate called SO.pem containing public key and
private key connecting to Load balancer

openssl s_client -key test_SO.pem -cert SO.pem -showcerts -connect
c2.company.net:443 -state -debug
```

7) How to convert pem and priv files to a JKS keystore?

We use PKCS 12 for this task

How to create Keystore (in 2 steps)? First I convert to PKCS12 then from PKCS12 to JKS

Create PKCS12 keystore from private key and public certificate.

```
openssl pkcs12 -export -name myservercert -in SO.pem -inkey
unencrypted_SO.priv -out SO.p12
```

Enter Export Password: (**** I entered 'Pa55w0rd' *****)
Verifying - Enter Export Password: (**** I entered 'Pa55w0rd' *****)

```
/usr/jdk1.8.0_60/bin/keytool -importkeystore -destkeystore SO.jks -
srckeystore SO.p12 -srcstoretype pkcs12 -alias myservercert
```

Enter destination keystore password: (**** I entered 'Pa55w0rd' *****)
Re-enter new password: (**** I entered 'Pa55w0rd' *****)
Enter source keystore password: (**** I entered 'Pa55w0rd' as I entered in the previous command line which creating PKCS12 *****)

How to verify the Keystore is created?

```
/usr/jdk1.8.0_60/bin/keytool -list -v -keystore Pa55w0rd.jks
```

Enter keystore password: (**** I entered 'Pa55w0rd' *****)

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: myservercert
Creation date: May 31, 2016
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=1135420, OU=Some Work, O=Company Inc., L=Bethesda, ST=Maryland, C=US
Issuer: CN=Company Certificate Authority, OU=Some Work, O="Company, Inc.", L=Arlington, ST=Virginia, C=US
Serial number: 11d734d9b5f

Valid from: Sat Mar 01 00:00:00 EST 2003 until: Thu Mar 01 00:00:00 EST 2103
Certificate fingerprints:
    MD5: 35:8E:AB:F9:DA:84:4B:6C:E2:5E:

C7:38:CA:49:67:2D
    SHA1: 7D:37:D0:DC:53:07:73:91:B8:5E:F8:5B:9F:2A:5A:5E:B2:64:A3:C1
    SHA256:
E5:70:F8:72:C2:E8:9C:BA:A6:70:6C:25:44:D0:19:E9:79:05:26:AC:3A:61:2C:35:4D:81:3A:F8:45:8D:4F:EA
    Signature algorithm name: SHA1withDSA
    Version: 3

How do I convert from JKS keystore to PEM?

```
/usr/jdk1.8.0_60/bin/keytool -export -alias companyca -keystore
companyCA.keystore -file companyCA.keystore.der.crt
Enter keystore password:  <----------------- (Enter the password from the
panel manufacturing server properties file)
Certificate stored in file <companyCA.keystore.der.crt>

openssl x509 -noout -text -in companyCA.keystore.der.crt -inform der

Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 1048611917 (0x3e808c4d)
    Signature Algorithm: dsaWithSHA1
        Issuer: C=US, ST=Virginia, L=Arlington, O=Company, Inc., OU=Some
Work, CN=Company Certificate Authority
        Validity
            Not Before: Mar 25 17:05:17 2003 GMT
            Not After : Mar 15 17:05:17 2043 GMT
        Subject: C=US, ST=Virginia, L=Arlington, O=Company, Inc., OU=Some
Work, CN=Company Certificate Authority
        Subject Public Key Info:
            Public Key Algorithm: dsaEncryption
                pub:
                    12:dc:40:de:57:3b:69:14:78:91:68:87:e2:dd:e0:
                    20:d3:ae:6b:6f:d2:04:ba:e6:bb:8a:16:53:88:28:
                    a2:a2:34:b2:31:d3:06:02:5a:19:6c:76:e4:00:43:
                    a6:21:92:67:31:83:6b:e7:cb:cc:53:51:e2:9b:b1:
                    d2:88:61:84:79:bc:0f:47:ba:45:5d:da:2f:2e:18:
                    b0:67:22:18:4a:cf:4e:fd:65:ff:ad:20:a6:db:08:
                    3c:bb:e5:32:08:7f:ea:51:27:ec:5f:a6:11:f5:16:
                    94:34:aa:c2:39:66:32:5d:bc:1c:2f:b6:22:94:98:
                    3a:09:7c:b5:79:82:2b:c7
                P:
                    00:fd:7f:53:81:1d:75:12:29:52:df:4a:9c:2e:ec:
                    e4:e7:f6:11:b7:52:3c:ef:44:00:c3:1e:3f:80:b6:
                    51:26:69:45:5d:40:22:51:fb:59:3d:8d:58:fa:bf:
                    c5:f5:ba:30:f6:cb:9b:55:6c:d7:81:3b:80:1d:34:
                    6f:f2:66:60:b7:6b:99:50:a5:a4:9f:9f:e8:04:7b:
                    10:22:c2:4f:bb:a9:d7:fe:b7:c6:1b:f8:3b:57:e7:
                    c6:a8:a6:15:0f:04:fb:83:f6:d3:c5:1e:c3:02:35:
```

```
                    54:13:5a:16:91:32:f6:75:f3:ae:2b:61:d7:2a:ef:
                    f2:22:03:19:9d:d1:48:01:c7
                Q:
                    00:97:60:50:8f:15:23:0b:cc:b2:92:b9:82:a2:eb:
                    84:0b:f0:58:1c:f5
                G:
                    00:f7:e1:a0:85:d6:9b:3d:de:cb:bc:ab:5c:36:b8:
                    57:b9:79:94:af:bb:fa:3a:ea:82:f9:57:4c:0b:3d:
                    07:82:67:51:59:57:8e:ba:d4:59:4f:e6:71:07:10:
                    81:80:b4:49:16:71:23:e8:4c:28:16:13:b7:cf:09:
                    32:8c:c8:a6:e1:3c:16:7a:8b:54:7c:8d:28:e0:a3:
                    ae:1e:2b:b3:a6:75:91:6e:a3:7f:0b:fa:21:35:62:
                    f1:fb:62:7a:01:24:3b:cc:a4:f1:be:a8:51:90:89:
                    a8:83:df:e1:5a:e5:9f:06:92:8b:66:5e:80:7b:55:
                    25:64:01:4c:3b:fe:cf:49:2a
    Signature Algorithm: dsaWithSHA1
         r:
             1d:ea:82:2a:59:89:9b:85:92:88:32:b4:2e:0d:b3:
             ea:d8:8b:70:f7
         s:
             74:e8:87:d0:fb:3d:ba:50:48:fb:67:af:e2:02:b0:
             38:ed:0c:68:0d
openssl x509 -in companyCA.keystore.der.crt -out companyCA.keystore.pem.crt -
outform pem -inform der

#to convert from JKS to PKCS12
/usr/jdk1.8.0_60/bin/keytool -importkeystore -srckeystore companyCA.keystore
-srcstoretype JKS -deststoretype PKCS12 -destkeystore companyCAkeystore.p12
```

How do I create a JKS keystore file directly using keytool?

```
/usr/java/jdk1.7.0_79/bin/keytool -keystore companyCA2016.keystore -
genkeypair -keyalg RSA -keysize 4096 -sigalg SHA512withRSA -storetype jks -
alias companyca
```

How to verify if the certificate fields for Subject and Issuer are encoded as PRINTABLESTRING or UTF8?

```
 /usr/jdk1.8.0_60/bin/keytool -export -alias companyca -keystore
companyCA2016.keystore -file companyCA2016.keystore.der.crt

Enter keystore password:
< Enter the company.ca.keyPasswordv2 >
Certificate stored in file <companyCA2016.keystore.der.crt>

 openssl x509 -in companyCA2016.keystore.der.crt -out
companyCA2016.keystore.pem.crt -outform pem -inform der

 openssl x509 -subject -issuer -subject_hash -issuer_hash -noout -nameopt
multiline,show_type -in companyCA2016.keystore.pem.crt

subject=
    countryName               = PRINTABLESTRING:US
    stateOrProvinceName       = PRINTABLESTRING:Maryland
    localityName              = PRINTABLESTRING:Bethesda
```

```
    organizationName          = PRINTABLESTRING:Company Inc.
    organizationalUnitName    = PRINTABLESTRING:Some Work
    commonName                = PRINTABLESTRING:Company Certificate Authority
issuer=
    countryName               = PRINTABLESTRING:US
    stateOrProvinceName       = PRINTABLESTRING:Maryland
    localityName              = PRINTABLESTRING:Bethesda
    organizationName          = PRINTABLESTRING:Company Inc.
    organizationalUnitName    = PRINTABLESTRING:Some Work
    commonName                = PRINTABLESTRING:Company Certificate Authority
9bacd77d
9bacd77d
```

How do I convert a JKS keystore to PKCS keystore?

```
/usr/jdk1.8.0_60/bin/keytool -importkeystore -srckeystore
companyCA2016.keystore -srcstoretype JKS -deststoretype PKCS12 -destkeystore
companyCA2016keystore.p12
```

How to convert a JKS keystore to pem and priv files?

First convert JKS to PKCS12 as above then do the following

```
# Public certificate
openssl pkcs12 -in companyCA2016keystore.p12 -clcerts -nokeys -nodes -out
companyCA2016.keystore.pub

# Private key encrypted
openssl pkcs12 -in companyCA2016keystore.p12 -nocerts -out
companyCA2016.keystore.priv.encrypt

# Private key decrypted
openssl pkcs12 -in companyCA2016keystore.p12 -nocerts -nodes -out
companyCA2016.keystore.priv.decrypt
```

How to convert a random RSA key pair to Certificate using the private key of an intermediate/root CA?

```
#Create Public key and Private key pair
#Generate un-encrypted key of length 1024
openssl genrsa -out test_key.pem 1024

#Extract private key
openssl rsa -in test_key.pem -pubout > test_key.priv

#Extract public key
openssl rsa -in test_key.pem -pubout > test_key.pub

#Generate CSR from the key pair (give both, the public key and the private
key as an input)
openssl req -new -in test_key.pub -key test_key.pem -out test_cert.csr

#Convert JKS to PKCS12
```

```
/usr/jdk1.8.0_60/bin/keytool -importkeystore -srckeystore companyCA.keystore
-srcstoretype JKS -deststoretype PKCS12 -destkeystore companyCAkeystore.p12

#Now extract Intermediate CA public key (optional)
openssl pkcs12 -in companyCAkeystore.p12 -clcerts -nokeys -nodes -out
companyCAkeystore.pub

#Now extract Intermediate CA private key (encrypted)
openssl pkcs12 -in companyCAkeystore.p12 -nocerts -out
companyCAkeystore.priv.encrypt

#Now extract Intermediate CA public key (decrypted)
openssl pkcs12 -in companyCAkeystore.p12 -nocerts -nodes -out
companyCAkeystore.priv.decrypt

#Generate cert using the earlier certificate signing request and private key
(signing key) of the intermediate CA from above
openssl x509 -req -days 365 -in test_cert.csr -signkey
companyCAkeystore.priv.decrypt -out test_cert.crt

#Convert crt to pem format
openssl x509  -in  test_cert.crt -out test_cert_output.pem

#To view public key from the certificate we have
openssl x509 -pubkey -noout -in companyCAkeystore.pub

#One may also verify CSR information as follows

#Extract information from CSR
openssl req -in test_cert.csr -text -noout

#Certificate issued to?
openssl req -in test_cert.csr -noout -subject

#verify the signature
openssl req -in test_cert.csr -noout -verify

#Show the public key
openssl req -in test_cert.csr -noout -pubkey
```

To view the contents of the private key

```
openssl rsa -in CP12615700.data.priv -text
Enter pass phrase for CP12615700.data.priv:
Private-Key: (1024 bit)
modulus:
    00:84:40:76:99:44:60:ec:7b:11:fd:3f:e6:95:c8:
    9b:47:5e:f2:0e:cb:06:5f:fb:fa:9d:52:19:3a:59:
    19:a3:1b:7c:b6:f5:5d:1a:8e:2a:99:45:75:33:e5:
    5e:9d:f3:96:e2:05:ed:28:6f:0b:52:f6:06:56:8f:
    ad:1b:96:5d:6c:7c:4c:0d:ab:84:05:29:71:14:f7:
    db:8d:fc:50:4c:77:5f:bf:e2:d8:cc:cd:88:a5:54:
    9a:6b:c5:6b:0d:7c:8a:54:98:20:31:86:8c:ff:6d:
    25:93:f2:4f:b5:ec:33:99:9e:95:86:85:db:3a:d1:
    6a:51:c9:63:c3:10:ba:93:45
publicExponent: 65537 (0x10001)
```

```
privateExponent:
    63:2e:ed:43:18:d3:0f:c7:64:c0:67:32:09:57:3f:
    8d:11:19:bc:1a:6b:17:85:24:78:e3:df:63:b0:fa:
    d7:26:80:2b:be:6c:2a:c4:40:12:5e:d2:fd:2e:a1:
    fd:17:78:2a:de:82:f3:f6:03:aa:1e:34:b6:aa:5e:
    0a:f8:83:eb:09:55:23:a9:f0:3c:a0:1b:8a:12:22:
    66:a7:24:96:02:66:68:6d:46:8d:9b:f6:0c:1f:c0:
    80:11:10:00:80:8f:eb:5c:7b:e7:2a:42:2c:16:33:
    b0:b9:a3:84:01:27:38:d7:2a:66:77:d6:96:36:7b:
    0a:c4:de:3f:cc:e2:94:e9
prime1:
    00:c9:74:2a:b3:4c:05:84:04:49:d2:96:03:4a:28:
    58:27:1b:8a:a5:25:8f:89:37:63:26:e7:1e:78:f6:
    87:71:82:ac:13:03:b5:28:30:88:65:d2:94:67:7f:
    9b:11:65:a0:76:29:23:a3:c3:ad:1d:d4:e1:c5:a4:
    52:af:75:19:fb
prime2:
    00:a8:0f:8a:38:86:22:5f:fe:84:6c:f8:53:82:30:
    78:32:fe:03:68:45:0a:f0:70:d7:ba:31:44:2f:9c:
    21:b3:e6:30:45:fc:cf:2c:57:7e:84:3c:98:bd:4d:
    a4:a3:59:17:f2:e4:d7:4a:fc:92:af:8b:b8:e9:f5:
    d0:0e:16:c3:bf
exponent1:
    00:c6:ab:38:5d:1f:d0:a4:b3:f1:f5:aa:99:4d:ed:
    e1:99:97:b0:b0:53:0d:6d:bd:e0:9b:81:fb:ec:40:
    fb:ab:b6:b9:69:fb:13:11:b2:63:21:3f:7d:b9:5f:
    ff:96:35:89:fc:5d:6f:d7:55:e7:08:ef:63:44:1d:
    5c:a9:ca:2b:37
exponent2:
    00:87:8d:61:33:0e:1d:93:c7:ba:10:1a:8b:60:aa:
    af:88:91:91:35:fc:da:41:ed:02:53:4f:81:6b:d5:
    46:e9:cf:74:88:3e:cc:eb:3a:f6:f4:b9:db:88:74:
    a3:a2:8d:2d:df:df:ec:36:b3:59:ac:f5:c9:84:0b:
    9c:70:80:b4:09
coefficient:
    00:ab:83:92:91:92:2a:0b:2e:f9:ef:a5:4a:62:12:
    5f:a3:b2:99:87:2e:b8:39:73:99:d4:3a:d7:11:05:
    8b:f8:62:85:37:84:f2:d9:60:90:97:98:eb:b1:00:
    07:fb:8d:a6:be:27:23:30:fe:1e:e2:d6:3e:b4:62:
    66:cd:f6:e3:8e
writing RSA key
```

-----BEGIN RSA PRIVATE KEY-----
MIICXgIBAAKBgQCEQHaZRGDsexH9P+aVyJtHXvIOywZf+/qdUhk6WRmjG3y29V0a
jiqZRXUz5V6d85biBe0obwtS9gZWj60bll1sfEwNq4QFKXEU99uN/FBMd1+/4tjM
zYilVJprxWsNfIpUmCAxhoz/bSWT8k+17DOZnpWGhds60WpRyWPDELqTRQIDAQAB
AoGAYy7tQxjTD8dkwGcyCVc/jREZvBprF4UkeOPfY7D61yaAK75sKsRAEl7S/S6h
/Rd4Kt6C8/YDqh40tqpeCviD6wlVI6nwPKAbihIiZqcklgJmaG1GjZv2DB/AgBEQ
AICP61x75ypCLBYzsLmjhAEnONcqZnfWljZ7CsTeP8zilOkCQQDJdCqzTAWEBEnS
lgNKKFgnG4qlJY+JN2Mm5x549odxgqwTA7UoMIhl0pRnf5sRZaB2KSOjw60d1OHF
pFKvdRn7AkEAqA+KOIYiX/6EbPhTgjB4Mv4DaEUK8HDXujFEL5whs+YwRfzPLFd+
hDyYvU2ko1kX8uTXSvySr4u46fXQDhbDvwJBAMarOF0f0KSz8fWqmU3t4ZmXsLBT
DW294JuB++xA+6u2uWn7ExGyYyE/fblf/5Y1ifxdb9dV5wjvY0QdXKnKKzcCQQCH
jWEzDh2Tx7oQGotgqq+IkZE1/NpB7QJTT4Fr1Ubpz3SIPszrOvb0uduIdKOijS3f
3+w2s1ms9cmEC5xwgLQJAkEAq4OSkZIqCy7576VKYhJfo7KZhy64OXOZ1DrXEQWL
+GKFN4Ty2WCQl5jrsQAH+42mvicjMP4e4tY+tGJmzfbjjg==
-----END RSA PRIVATE KEY-----

References:

1. http://www.akadia.com/services/ssh_test_certificate.html
2. https://en.wikipedia.org/wiki/Certificate_authority
3. https://www.blackmoreops.com/2015/05/12/ssl-sign-with-sha256-hash-using-openssl/
4. https://myonlineusb.wordpress.com/2011/06/19/what-are-the-differences-between-pem-der-p7bpkcs7-pfxpkcs12-certificates/
5. https://blog.didierstevens.com/2013/05/08/howto-make-your-own-cert-and-revocation-list-with-openssl/
6. http://openssl.6102.n7.nabble.com/create-certificate-chain-td44046.html
7. https://www.madboa.com/geek/openssl/
8. http://www.shellhacks.com/en/HowTo-Decode-CSR