SSL Public Key Certificates
Overcoming the man-in-the-middle attack
Summary and Hands-on Guide
peter-thoemmes.org research

© Peter Thoemmes
Weinbergstrasse 3a
D-54441 Ockfen, Germany

December 23, 2011

Abstract

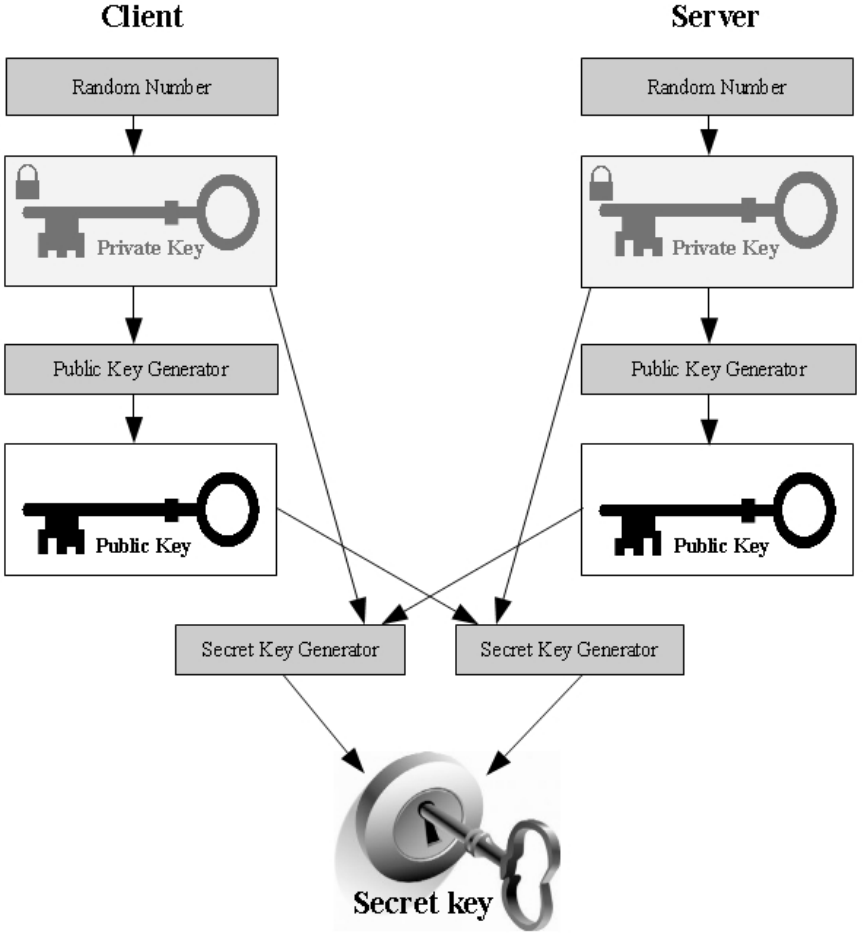
This paper is a summary and a hands-on guide for technical people, who like to know how SSL public key certificates really work. The aim is to show the motivation and the complete story behind the idea of a Public Key Infrastructure (PKI) using SSL public key certificates. This paper is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY, without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Contents

1 General Stuff

Before the idea of SSL public key certificates, there was the idea of a key agreement using an insecure connection. That idea of Diffie and Hellmann was almost perfect. Never the actual key was transferred, but still after the initial handshake both parties knew which secret encryption key to use.

Figure 1: Diffie-Hellmann Key Agreement Method



But that method had and has a weak part: it is possible that a hacker (user C) hooks into the traffic of client (user A) and server (user B). When A initiates a conversation, C fetches all messages and pretends to be B. Knowing the key agreement method, C will send back a public key part, pretending to by B, while spoofing the address of B, and so A will think that this is user B at the other end of the line and finally will setup a proper encrypted connection to C. So C will get all the information meant for B and C will be able to fully decrypt it. This is known as the man-in-the-middle attack.

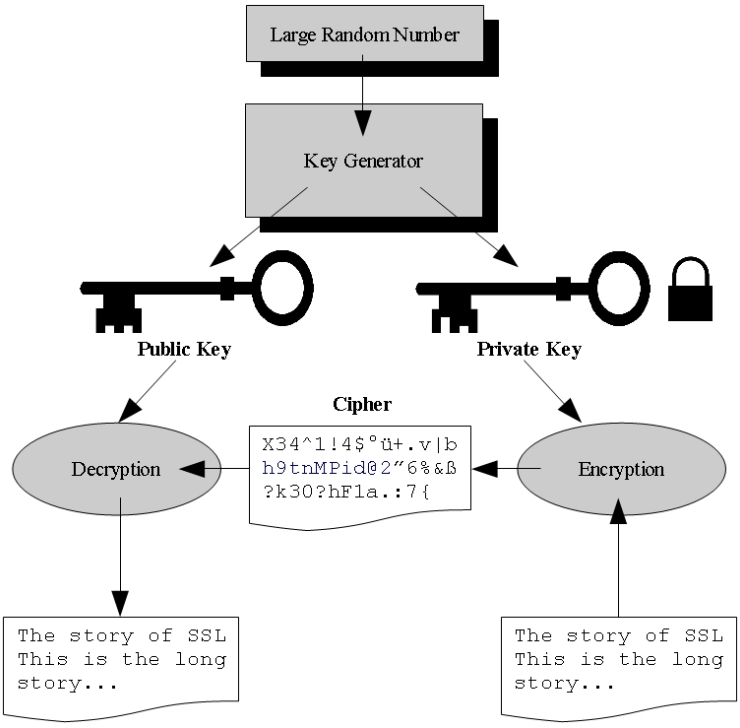
To overcome the man-in-the-middle attack, the mechanism of sending the fingerprint of the public part of the RSA host key was introduced. So before the actual encrypted connection was setup, the user was able to verify the correctness of the identity of the other user by cross-checking the fingerprint. To do so, any kind of manual transmission of the authentic fingerprint is required before, possibly by a

phone call to the other user. That is a fairly safe method, but time-consuming and of course it needs to be understood by the user. Typically the remote user is a server, which is maintained by an administrator, who might be able to pick up the phone and read out the server's RSA host key fingerprint. If the server runs OpenSSL, he can read the fingerprint like this:

```
$ ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key.pub
2048 c3:f6:2a:01:cd:39:61:7f:df:53:57:3e:d8:e4:99:36 /etc/...st_rsa_key.pub
```

As many users do not understand that fingerprint thing, and as it is way to complicated if many servers need to be contacted using a secured encrypted line, a new and fully automated method was developed: **PKI** (Public Key Infrastructure). PKI is an automated solution to overcome the man-in-the-middle attack problem. Using that method a server provides its public key as it did for the Diffie-Hellmann method, but it does so in a signed certificate so-called **SSL public key certificate**. That is an electronic document which uses a digital signature to bind a public key with an identity. The signature in a certificate is an attestation by the certificate signer (issuer) that the identity information and the public key belong together. A client (e.g. a web-browser) then verifies the signature of that certificate by the signature decryption key of the signer (issuer), who is called Certificate Authority (CA) in a PKI. The signature decryption key is another public key and is not to mess up with the signed public key inside the certificate. The issuer (CA) did encrypt the signature with its private key of an **asymmetric** key pair. The client can decrypted the signature with the public key of that asymmetric key pair. That's the way an asymmetric key pair works.

Figure 2: Asymmetric Encryption Keys



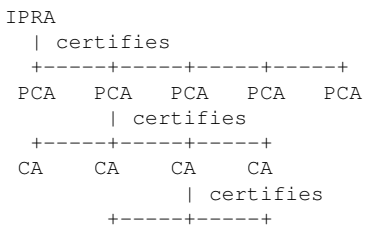
So a client simply needs to know the public key of the issuer (CA) to be able to decrypt it. To enable all clients to do so, the deployment of the public (signature

decryption) key of the root instance of a PKI is done during the installation of the encryption software (e.g. OpenSSL or Firefox web-browser). For the Internet community that root instance is the IPRA (Internet Policy Registration Authority). A client can now verify the server's public key by a certificate, signed by a CA. He can further verify the CA's public (signature decryption) key by the CA's certificate and so on. That loop goes up until a certificate is self-signed, meaning the issuer (signer) is at the same time the holder of the public key inside. Then the root CA is reached, e.g. the IPRA. That's the way SSL clients are verifying public keys provided by a server.

Although the man-in-the-middle attack can be overcome by PKI, there is now another problem: it requires essentially trusting the instances that deploy certificates. If such an instance signs a faked certificate, then the method fails totally. So all the PKI principle is fully based on trust.

2 How is a Public Key Infrastructure setup?

To sign SSL public key certificates Certificate Authorities (CA) are setup. The IPRA (Internet Policy Registration Authority) acts as the root of the certification hierarchy for the Internet community. The IPRA signs the certificates of all PCAs (Policy Certification Authorities), the PCAs sign the certificates of the CAs and the CAs sign certificates of users (servers).

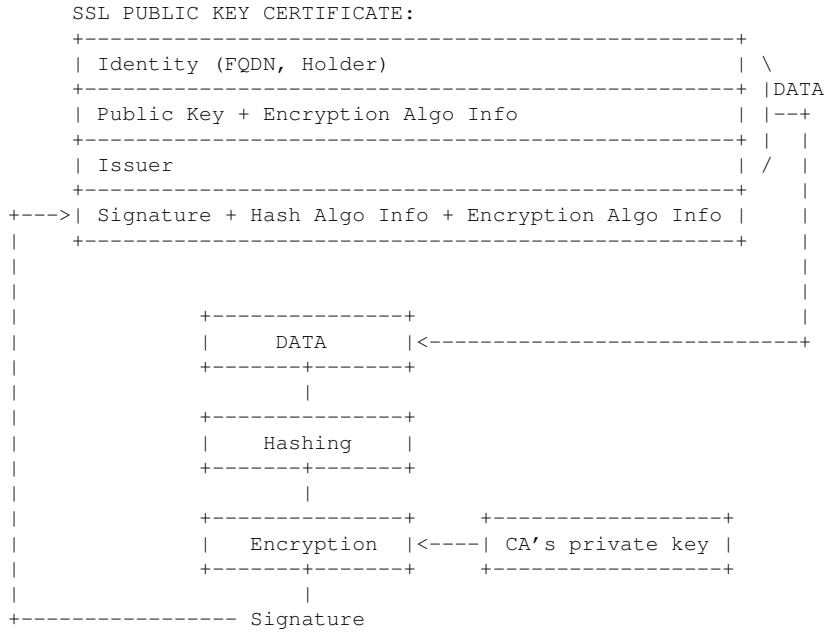


The public (signature decryption) key to validate certificates signed by the IPRA is made available by installing any SSL client software. So an SSL client will verify the signature of the certificate provided by a contacted server, which is signed by a CA. Then it will verify the CA's certificate, which is signed by a PCA, and so on. It will break that loop if it gets on to a certificate that is self-signed, meaning the issuer is at the same time the holder of the public key inside. If that self-signed certificate is installed as 'trusted', the holder will be seen as the root CA and the validation chain is completed. If not so, the validation chain is not completed and so the server's SSL public key certificate will not be trusted, as it has no trusted signer. A special case is a server certificate, that is self-signed. Self-signed server certificates should only be used in experimental environments. Local caching of already validated certificates can be done and will speed up the validation process significantly. If local caching is implemented, then this is called a User Authority (UA).

3 How is the certificate created and verified?

Certificate issuer (CA, Certification Authority):

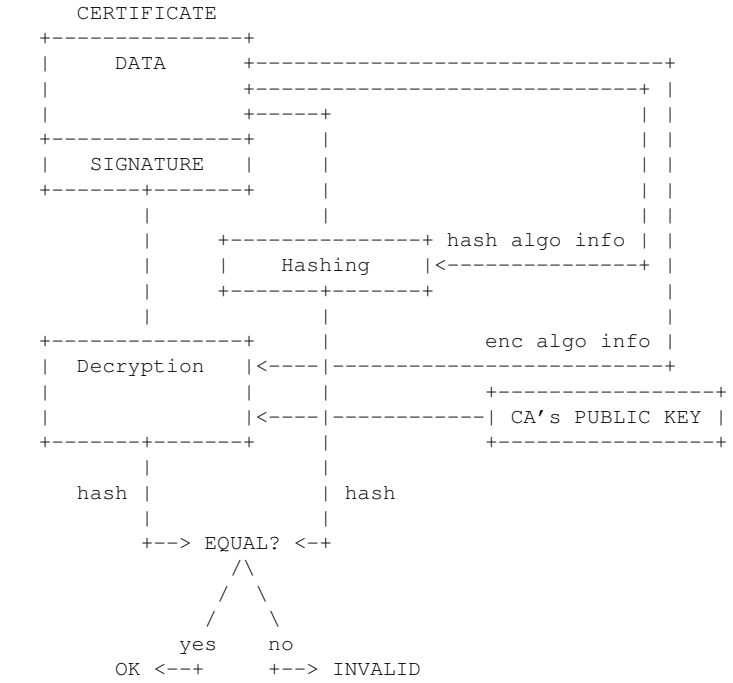
The CA (issuer) gets contacted by a server's administrator, who wants to have a certificate signed. He tells the server's public key and the fully qualified domain name (FQDN). The CA ensures that the binding is unique and that the server behind the FQDN is really mapped to an IP address (DNS) that belongs to the company running the server (holder). So a full identity record (FQDN and holder) is build and the public key and the issuer and a few more information is appended. All together that is called the **data** of the certificate. That data is pushed through an hashing algorithm. The resulting hash then is encrypted using the private (signature encryption) key of the CA's **asymmetric** key pair. The resulting cipher is the **signature** for that data. Finally the full blown and signed certificate to the server administrator's request is built:



That SSL public key certificate is send back to the administrator for installation on the server machine.

Certificate user (SSL client):

The user, after he got presented a certificate by the server, will do a validation. To do so he reads from the certificate what is the hash algorithm used for creating the signature (hash algo info, e.g. 'sha1') and then he does the same as the CA already did, meaning he hashes the data. Using the CA's public (signature decryption) key and the certificate's info about how the signature is encrypted (encryption algo info, e.g. 'RSA'), he decrypts the signature from the certificate and checks if the result is equal to the calculated hash. If yes, the certificate is valid, meaning the public key inside can be used to encrypt a message to the server behind the linked FQDN. For doing so, the encryption algo info linked to the public key is to regard, meaning the correct encryption algorithm (e.g. 'RSA') is to use.



All this only works, if the CA's public (signature decryption) key can be trusted. This is automatically the case if it is locally installed as 'trusted' (local caching). If not locally installed, it needs to be validated first. This is done the same way a server's certificate is validated, as a CA's public (signature decryption) key is also wrapped into a certificate. Doing so, the SSL client climbs up the PKI tree until it finds a self-signed certificate, that is locally installed as 'trusted' (= root CA). If the validation chain is not ending at a trusted root CA, the server's certificate will not be verified using the CA's public key, but it will be seen immediately as untrusted, as the signer is not trusted.

4 How is the technical implementation done?

When an SSL client connects to an SSL server, the server presents a certificate, so essentially an electronic piece of proof that the server is, who it claims to be. This certificate is signed by a 'Certificate Authority' (CA), usually a trusted third party like 'VeriSign'. The client hashes the data inside with the hash algorithm it knows from the 'hash algo info'. Then it checks out if it has installed a certificate under the name '*hash*.0'. Installing a validated certificate under '*hash*.0' is the way local caching of validated certificates is done. If '*hash*.0' is found, the client compares that file's content with the certificate provided by the server. If both are equal the certificate is valid. If '*hash*.0' is not found, it decrypts the signature by help of the 'encryption algo info' and the signature decryption key, which is the issuer's (CA's) public key. If the result is equal to the calculated hash, the certificate is valid, as long as the CA's public (signature decryption) key is trusted (meaning the CA's public key is installed in an appropriate '*hash*.0' file or it can be validated climbing up the PKI tree). If the certificate is valid (pre-installed or signature ok) and the current date is inside the **validity period** ('Validity' field inside the certificate) and the certificate is **not blacklisted**, then it is trusted. And then the client builds up an encrypted connection to the server (the linked 'encryption algo info' tells it the correct encryption algorithm) using a secret key made of his private and the server's public key (= Diffie-Hellmann Key Agreement Method).

REMARK:

> Not just SSL servers, but also SSL clients can present certificates. In that case those are called 'client certificate' or 'peer certificate'.

It is still possible to manually validate a certificate by contacting the server's administrator by a phone call and asking him about the MD5 fingerprint of the server's public key. To get the MD5 fingerprint of the public key included in a certificate, one can use following OpenSSL command (assuming the certificate is stored in the file 'server.pem'):

```
$ openssl x509 -md5 -noout -fingerprint -in server.pem
MD5 Fingerprint=88:D0:07:2E:59:0C:A9:74:3D:09:CA:32:1D:F9:A1:92
```

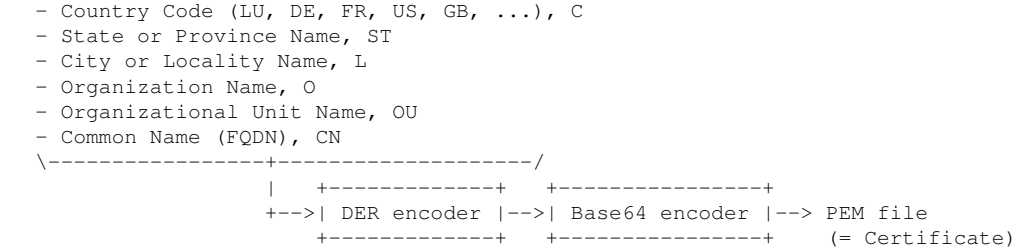
If the displayed value is equal to the one told by the server's administrator, the certificate can be stored as 'trusted', meaning under '*hash*.0'. The hash value can be determined by OpenSSL like this:

```
$ openssl x509 -subject_hash -noout -in server.pem
9031871d
```

5 Hands-on guide using Linux and OpenSSL

ITU-T X.509 defines a framework for public key certificates for Public Key Infrastructures (PKI) and ITU-T X.690 defines a set of Basic Encoding Rules (BER) for those certificates.

A certificate is stored in a format called DER (Distinguished Encoding Rules). DER is a subset of BER (Basic Encoding Rules) in a way that it eliminates all of the sender's options from DER. Typically after the conversion into that format a Base64 (RFC 3548) encoding is done, which leads to the final format, called PEM (Privacy Enhanced Mail):



The actual PEM payload is put in between 2 delimiters (2 lines which tell the beginning and the end of the certificate), e.g. in a file 'server.pem':

```
# cat server.pem
-----BEGIN CERTIFICATE-----
MIIDEzCCAnwCCQCrkMTpAcMqcTANBgkqhkiG9w0BAQUFADCBzTELMAkGA1UEBhMC
WFgxKjAoBgNVBAgTIVRoZXJlIGlzIG5vIHN1Y2ggdGhpbmcgb3V0c2lkZSBVUzET
MBEGA1UEBxMKRXZlcnl3aGVyZTEOMAwGA1UEChMFT0NPU0ExPDA6BgNVBAsTM09m
ZmljZSBmb3IgQ29tcGxpY2F0aW9uIG9mIE90aGVyd2lzZSBTaW1wbGUgQWZmYWly
czERMA8GA1UEAxMIbGFtcG1haW4xHDAaBgkqhkiG9w0BCQEWDXJvb3RAbGFtcG1h
aW4wHhcNMDkxMjEwMDkwNTA0WhcNMTAwMTA5MDkwNTA0WjCBzTELMAkGA1UEBhMC
WFgxKjAoBgNVBAgTIVRoZXJlIGlzIG5vIHN1Y2ggdGhpbmcgb3V0c2lkZSBVUzET
MBEGA1UEBxMKRXZlcnl3aGVyZTEOMAwGA1UEChMFT0NPU0ExPDA6BgNVBAsTM09m
ZmljZSBmb3IgQ29tcGxpY2F0aW9uIG9mIE90aGVyd2lzZSBTaW1wbGUgQWZmYWly
czERMA8GA1UEAxMIbGFtcG1haW4xHDAaBgkqhkiG9w0BCQEWDXJvb3RAbGFtcG1h
aW4wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAJfmq1A2ndYCaBg4IOCwemoH
u82tw82nzZcQ2jjIwkIiF4aTZTdYQ+zA6/vRtuSnHX4YBr2eix9aAqPo6LY6x5xR
lbpXjMMUwZWSnVDjJDBBgNJaK15T+6lO8P42gPyoyNCOHe0WtThdYdiWEMGloyUU
v+kMDDUVJLXP+fCAtU1NAgMBAAEwDQYJKoZIhvcNAQEFBQADgYEACG5eCgw/4wEj
LqG9DaKvLi5KMt9OKJ5K+FrvpD2w3ItNusgZUF/E7CSJ1zlA/I1hHnXMudzVdNbm
XaFMFP+3jwG2Icf2JpJU+7fDuZVMfZAXkln2oXtp1ONxJuAs8DAGOAXHhBiDKkag
cviDz2IdKilOd+vd6ffca33SX6w+dO4=
-----END CERTIFICATE-----
```

So this is all following exactly the ideas mentioned in the chapters above. The identity is not stored using the original DER format, but Base64 coded. This is not a secure encryption, as it can be decoded with any standard Base64 decoder (e.g. http://base64decode.org/), but is is a pure 7 bit ASCII representation of the certificate, that will be handled well by every text processor.

REMARKS:

1.) Other delimiter lines will also be handled well:

```
-----BEGIN X509 CERTIFICATE-----
-----END X509 CERTIFICATE-----

-----BEGIN TRUSTED CERTIFICATE-----
-----END TRUSTED CERTIFICATE-----
```

```
2.) To manually import an SSL certificate from a server, say 'host.com':

       # openssl s_client -connect host.com:443 | tee ./server.log
       ...
       QUIT<RETURN>

    Remove everything before the line 'BEGIN ... CERTIFICATE'.
    If after that the line 'END ... CERTIFICATE' there is Diffie-Hellman
    parameters appended ('-----BEGIN DH PARAMETERS-----'), then remove
    everything after the line '-----END DH PARAMETERS-----', else just
    remove everything after the line 'END ... CERTIFICATE':

       # cp ./server.log ./server.pem
       # vi ./server.pem
       -----BEGIN CERTIFICATE-----
       ...
       -----END CERTIFICATE-----
       -----BEGIN DH PARAMETERS-----      \
       ...                                |-> OPTIONAL
       -----END DH PARAMETERS-----        /


3.) To manually generate a most simple self-signed certificate 'server.pem'
    for the FQDN 'myhost.domain':

       # cd /etc/ssl/certs/
       # openssl req -new -x509 -days 365 -nodes \
       -out server.pem -keyout server.pem
       ...
       Country Name (2 letter code) [AU]:US<RETURN>
       State or Province Name (full name) [Some-State]:Nevada<RETURN>
       Locality Name (eg, city) []:<RETURN>
       Organization Name (eg, company) [Internet Widgits Pty Ltd]:<RETURN>
       Organizational Unit Name (eg, section) []:<RETURN>
       Common Name (eg, YOUR name) []:myhost.domain<RETURN>
       Email Address []:<RETURN>

    To append Diffie-Hellman parameters:

       # openssl gendh 512 >> server.pem
```

The DER format defines several data fields like

```
- Version: ...
- Serial Number: ...
- Signature Algorithm: ...
- Issuer: C=XX, ST=..., L=..., O=..., OU=..., CN=...      <== ISSUER's ID
- Validity: Not Before: ..., Not After: ...
- Subject: C=XX, ST=..., L=..., O=..., OU=..., CN=FQDN    <== HOLDER's ID
- Subject Public Key Info: ...                            <== HOLDER's PubKey
     Public Key Algorithm: rsaEncryption                  <== enc algo info
     RSA Public Key: (1024 bit)
         Modulus (1024 bit):
             00:97:e6:ab:50:36:9d:d6:02:68:18:38:20:e0:b0:
             7a:6a:07:bb:cd:ad:c3:cd:a7:cd:97:10:da:38:c8:
             c2:42:22:17:86:93:65:37:58:43:ec:c0:eb:fb:d1:
             b6:e4:a7:1d:7e:18:06:bd:9e:8b:1f:5a:02:a3:e8:
             e8:b6:3a:c7:9c:51:95:ba:57:8c:c3:14:c1:95:92:
             9d:50:e3:24:30:41:80:d2:5a:2b:5e:53:fb:a9:4e:
             f0:fe:36:80:fc:a8:c8:d0:8e:1d:ed:16:b5:38:5d:
             61:d8:96:10:c1:a5:a3:25:14:bf:e9:0c:0c:35:15:
             24:b5:cf:f9:f0:80:b5:4d:4d
         Exponent: 65537 (0x10001)
```

followed by the ISSUER's signature:

```
Signature Algorithm: sha1WithRSAEncryption       <== hash & enc algo info
  08:6e:5e:0a:0c:3f:e3:01:23:2e:a1:bd:0d:a2:af:2e:2e:4a:
  32:df:4e:28:9e:4a:f8:5a:ef:a4:3d:b0:dc:8b:4d:ba:c8:19:
  50:5f:c4:ec:24:89:d7:39:40:fc:8d:61:1e:75:cc:b9:dc:d5:
  74:d6:e6:5d:a1:4c:14:ff:b7:8f:01:b6:21:c7:f6:26:92:54:
  fb:b7:c3:b9:95:4c:7d:90:17:92:59:f6:a1:7b:69:d4:e3:71:
  26:e0:2c:f0:30:06:38:05:c7:84:18:83:2a:46:a0:72:f8:83:
  cf:62:1d:2a:29:4e:77:eb:dd:e9:f7:dc:6b:7d:d2:5f:ac:3e:
  74:ee
```

All this can be retrieved after decoding

```
               +-----------------+   +-------------+
PEM file ----->| Base64 decoder |-->| DER decoder |--> certificate text
(Certificate) +-----------------+   +-------------+
```

using following OpenSSL command:

```
$ openssl x509 -inform PEM -in server.pem -text -noout
```

To make OpenSSL doing a full blown verification of a certificate:

```
$ openssl verify server.pem
server.pem: OK
```

Another way to validate is the manual validation by the fingerprint. To get the MD5 fingerprint of the contained public key:

```
$ openssl x509 -md5 -noout -fingerprint -in server.pem
MD5 Fingerprint=88:D0:07:2E:59:0C:A9:74:3D:09:CA:32:1D:F9:A1:92
```

After you need to call the holder and ask him to tell you the fingerprint on the phone. For that he will need to do the same manual validation as you, but locally on the server machine.

```
!!! Only if you got the fingerprint from a trusted source !!!
!!! you can go ahead installing the certificate           !!!
```

If the certificate is a trusted one, then install it. This is done by calculating a hash over the subject field (this is the identification of the holder of the public key inside):

```
$ openssl x509 -subject_hash -noout -in server.pem
9031871d
```

Then the PEM file is copied to /etc/ssl/certs/

```
# cp ./server.pem /etc/ssl/certs/
```

and a symbolic link '*hash*.0' is created under /etc/ssl/certs/:

```
# openssl x509 -subject_hash -in server.pem -noout | tee /tmp/hash
# ln -s /etc/ssl/certs/server.pem /etc/ssl/certs/`cat /tmp/hash`.0
```

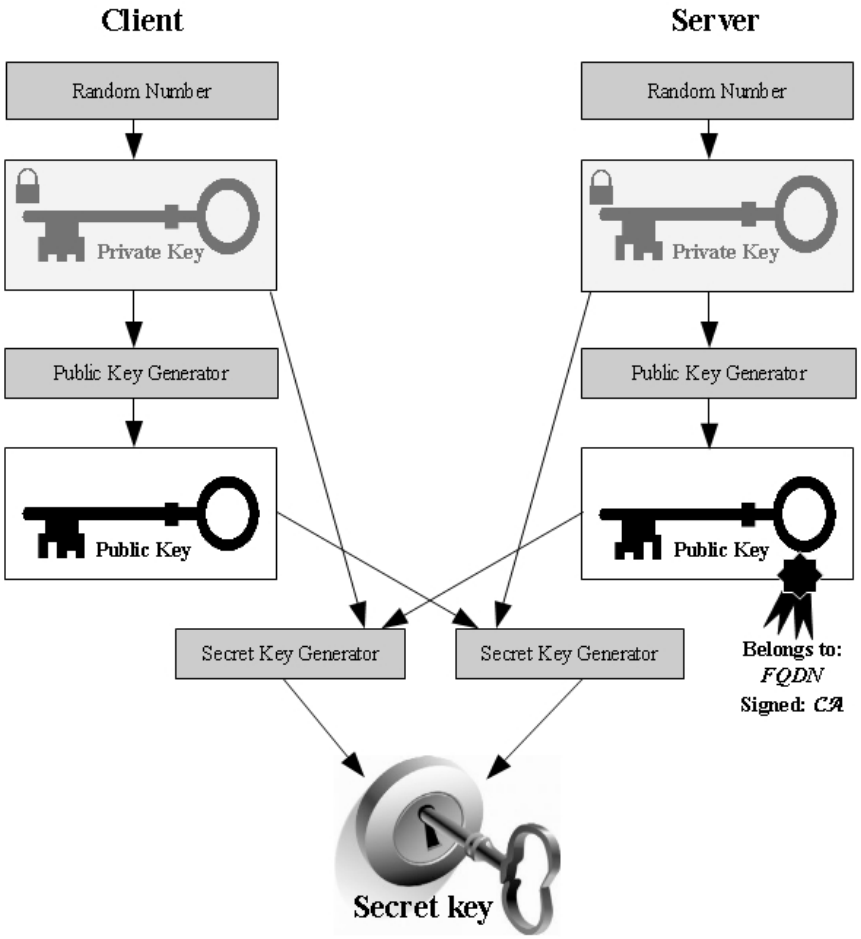
To be sure everything went fine, the installed certificates can be verified again:

```
$ openssl verify /etc/ssl/certs/`cat /tmp/hash`.0
```

6 Summary

After all the things mentioned above, it should be clear, that the SSL communication encryption is identical with the one used by SSH, SCP and SFTP. So all this methods use a secret key for encrypting/decrypting the data transmitted over the network, which is negotiated using the Diffie-Hellmann Key Agreement Method. The difference is, that **SSL patches an FQDN to the public key and gets it signed from a trusted third party**, a so-called Certificate Authority (CA). So SSL servers offer their public key wrapped into an **SSL public key certificate**.

Figure 3: SSL Keys and the Diffie-Hellmann Key Agreement Method



Be aware that **there is 2 kind of public keys in the game**. The **server's public keys** are wrapped into SSL public key certificates and they are used to build the secret encryption/decryption key for the network communication. The **CA's public keys** are also wrapped into public key certificates, but they are used to directly decrypt the signatures of certificates. So with those public keys a server's public key certificate or another CA's public key certificate can be validated.

Figure 4: SSL Public Key Certificate