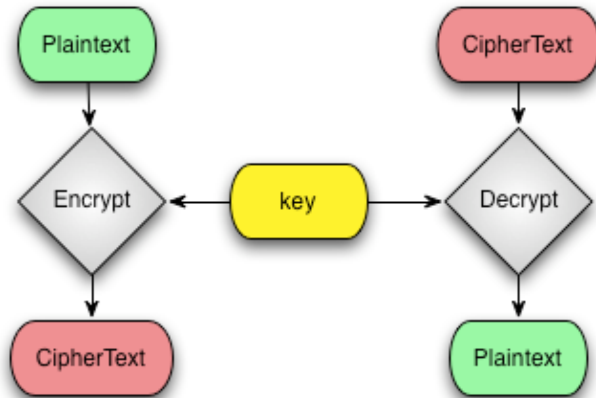
PKI, X.509 CERTIFICATES AND HTTPS WEBSERVERS

Public Key algorithms, digital certificates and PKI

PRELIMINARIES

Symmetric/Asymmetric cryptography



symmetric

The encryption and decryption keys are the same or can be directly derived from each other. Both keys are kept secret.
Examples: 3DES, AES, Blowfish, RC4

asymmetric

Encryption/decryption keys are different and it is computationally unfeasible to derive them from each other.
The encryption key be distributed, the other has to be kept secret.
For this reason it is also called Public Key cryptography.
Examples: RSA, Diffie-Hellman, ElGamal

Public Key cryptography: encryption/decryption

Alice

Alice wants to send a message *M* encrypted for Bob

Bob

Gets Bob's public key B_{pub}
(Somehow) verifies B_{pub} authenticity
Encrypts *M* with $B_{pub} \rightarrow C = F(B_{pub}, M)$

Alice sends *C* to Bob

Decrypts C with Bob's private key B_{priv}
$M = F(B_{priv}, C)$

Note:

1) Only Bob can decrypt *C*
2) Nobody "can" derive B_{priv} from B_{pub}
3) This procedure can be inverted to implement a digital signature

Public Key cryptography: digital signature

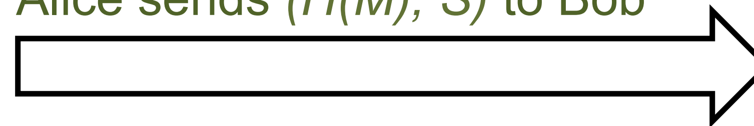
Alice

Alice wants to sign a message *M* so that Bob can verify its authenticity

Bob

Gets his own private key A_{priv}
Computes a hash of the message $H(M)$
Signs H(M) with $A_{priv} \rightarrow S = F(A_{priv}, H(M))$

Alice sends *(H(M), S)* to Bob



Computes a hash of the message $H(M)$
Verify the signature by verifying the following:
$H(M) = F(A_{pub}, H(M))$?

Note:

1) Only Alice can sign *M*
2) Nobody can modify M and compute a valid signature S without knowing A_{priv}
3) Alice can include a nonce (given by Bob) in the signature to avoid a third entity to reuse the same signature for the same message *M*

RSA: key generation

1. Extract two “big” prime numbers ***p*** e ***q*** **(random, secret)**
2. Compute the RSA modulus: $N = p \times q$
3. Compute $\Phi(N) = (p - 1)(q - 1)$ (Eulero’s function)
4. Randomly generates the the number **e:** $1 < e < \Phi(N)$ relatively prime to $\Phi(N)$
5. Compute the number **d:** $e \times d = 1 \bmod \Phi(N)$, or in other words **e** is the inverse of ***d*** in the group $\Phi(N)$

PUBLIC KEY: *(N, e)*

PRIVATE KEY: *d*

Must be kept secret: ***p***, ***q***, $\Phi(N)$, ***d***

Note:

1) *to derive d from e an attacker should compute* e^{-1} *in* $\Phi(N)$

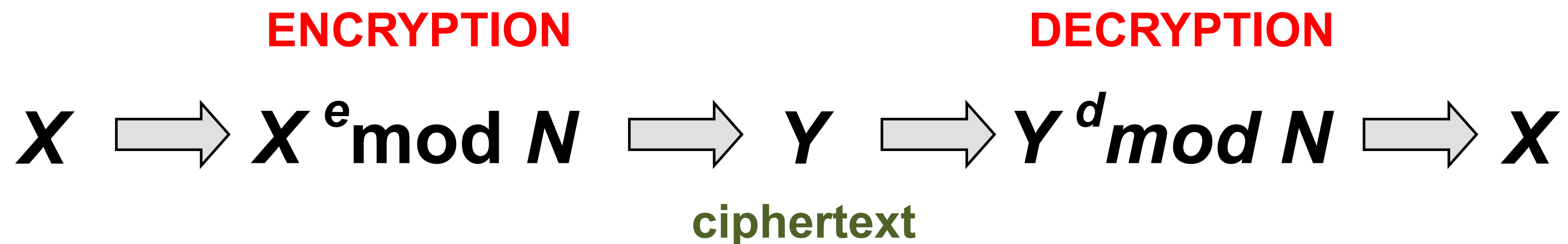
2) $\Phi(N)$ *is the number of integers less than or equal to n that are relatively prime to N*

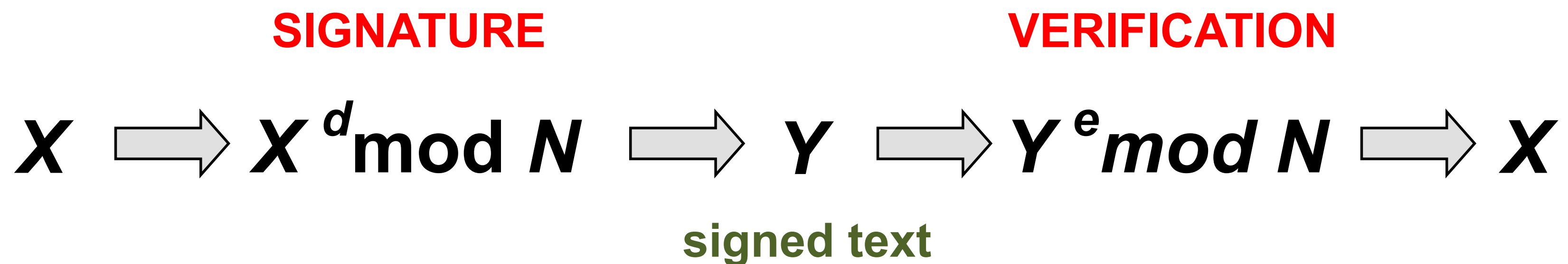
2.1) *to compute* $\Phi(N)$ *an attacker should know p and q (otherwise it’s unfeasible)*

3) *it is computationally unfeasible to factorize the product of two “big enough” prime numbers*

RSA tranformations

RSA transformation is simply a modular exponentiation with respectively the public private key





RSA with mathematica

…not <u>exactly</u> the real algorithm, but the concepts are the same!

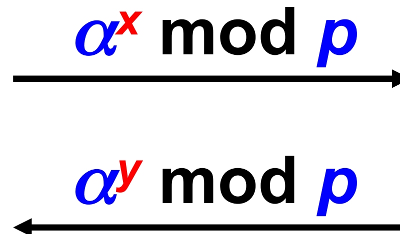
Diffie-Hellman Key exchange alogorithm

Public: α, p

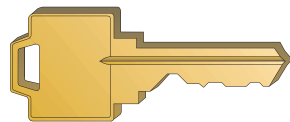
Secret: x, y

GOAL: exchange a common secret that only Alice and Bob can derive

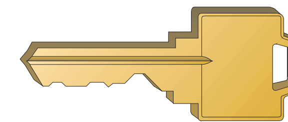
Random x



Random y



COMMON KEY



$$K = (\alpha^y)^x \bmod p$$

$$K = (\alpha^x)^y \bmod p$$

Note:

1) *Common secret number exchanged with an asymmetric algorithm*

2) *to compute K from (α^x mod p) and (α^y mod p) an attacker should be able to compute the discrete logarithm $x = \log_\alpha(\alpha^x \bmod p)$ and $y = \log_\alpha(\alpha^y \bmod p)$…*

3) *…which is computationally unfeasible for an attacker with "limited computational resources"*

How does Alice obtain Bob’s public key?

- Everything’s perfect, you believe that nobody can break the public key algorithms if the numbers are “big enough”
- How are the public keys distributed?
  - In a network with n nodes, n(n-1)/2 keys have to be distributed!
  - What if my private key is lost or stolen? Should I need to notify all the remaining (n-1) nodes to revoke my public key?
  - Solution: centralized or opportunistic distribution! (obvious, the public key don’t have to be kept secret!)
- OK, the scalability issue is solved, but how can I be sure that a public key is authentic? How can Alice get the public key of Bob and be sure that it’s really his?
- SOLUTION:
  - A trusted third party that issues some kind of proof that a public key is really related to a given identity

Public Key Certificate

- A public key certificate is a data structure that binds a public key (and therefor the related private key) to the the identy of the legitimate owner → $CERT_{ID}$:{ID, Pub_{ID}}
- The binding between {ID, Pub_{ID}} is granted by a trusted certification authority that signs $CERT_{ID}$
- Provided that we have the CA's public key, we can verify the CA signature and therefor verify the public key authenticity

EXAMPLE:

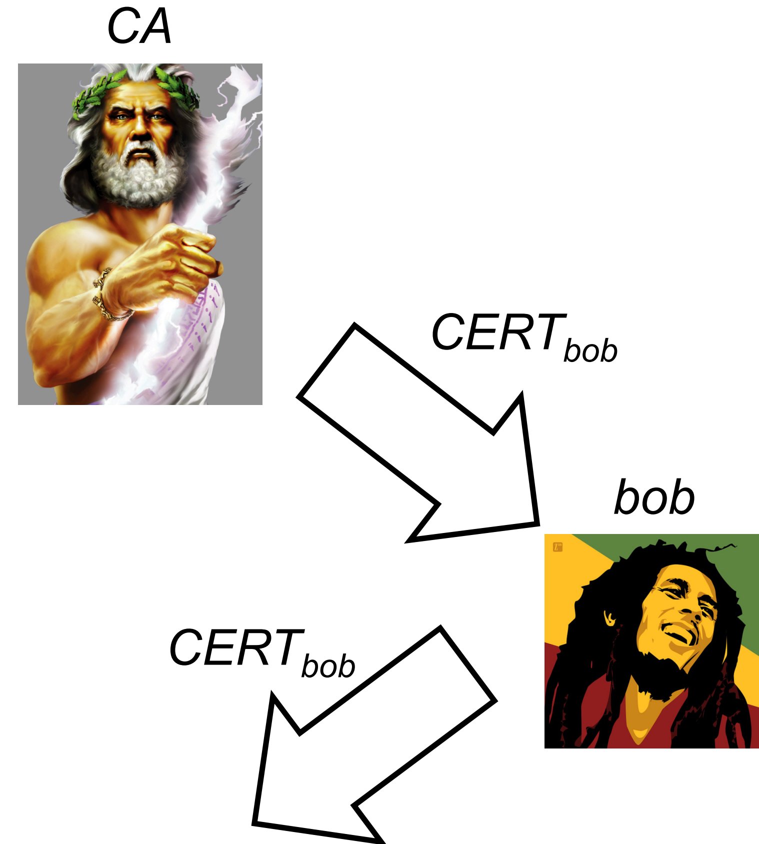
CA issues a public certificate for bob $CERT_{bob}$

$CERT_{bob}$ contains:

1) Pub_{bob}
2) CA identity CA_{id}
3) CA signature of $CERT_{bob}$

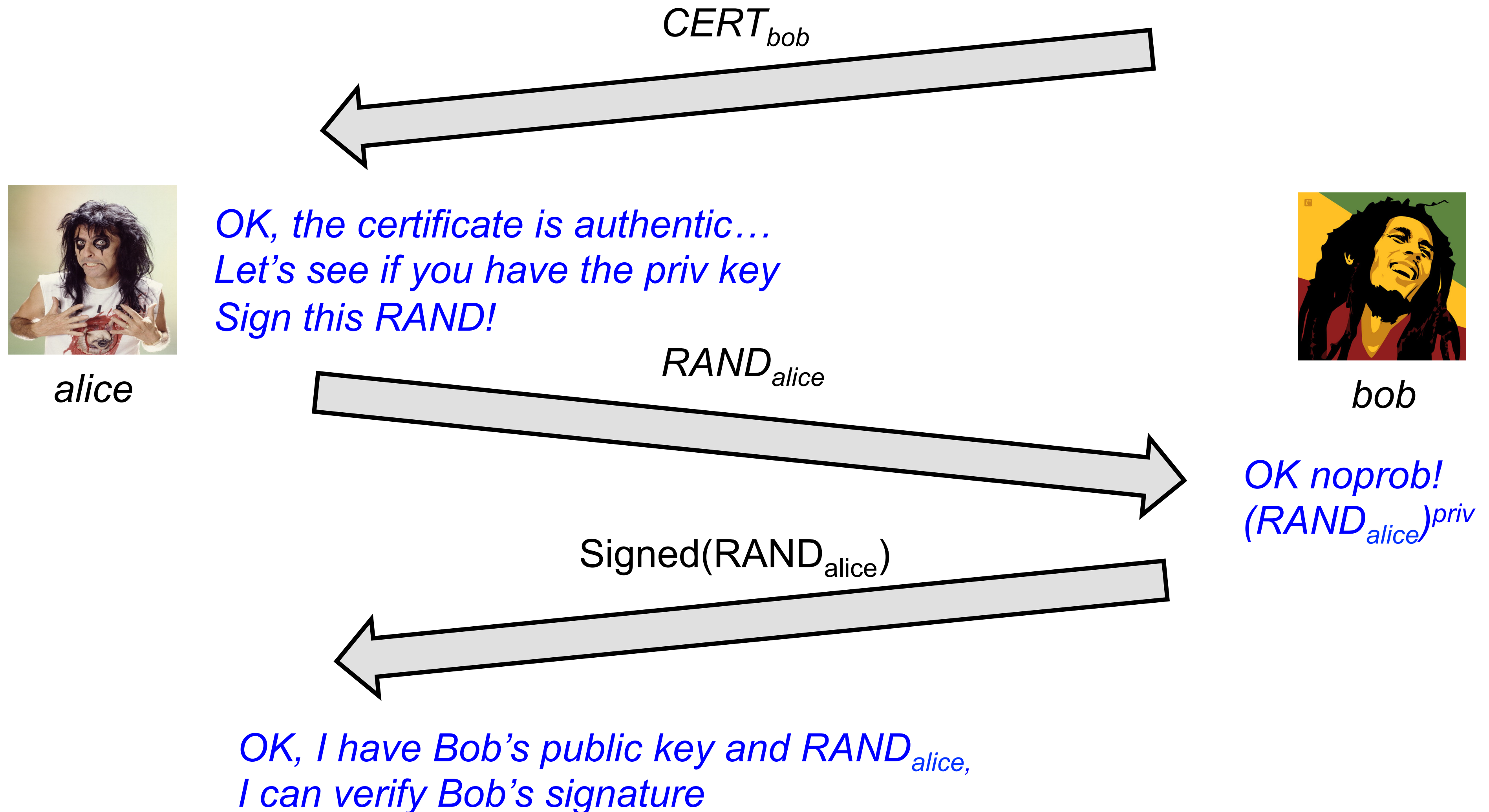
Once I have the authentic Pub_{bob}, I just need to verify that the party I'm communicating with is actually Bob (i.e.: it has the private key)

To do so, I perform a simple challenge/response mechanism. I extract a nonce and challenge Bob to sign this random number. Since the public key is authentic, and Bob couldn't know the random number, only the real Bob can sign the nonce correctly (and I can verify it)



- I trust CA and I have CA's public key
- Verify CA signature $CERT_{bob}$ → OK!
- Pub_{bob} is authentic
- I can encrypt a message for Bob

Challenge/Response concept



Public Key Infrastructure

- A PKI consists of the protocols, the policies and the cryptographic mechanism used to manage the management of public key certificate
  - Creation, distribution, revocation, etc…
- A PKI requires the definition of:
  - Certificate format
  - Relationship among CAs
  - Mechanisms and policies for issuing and revoking certificate
  - Storage services
- Typical certificate format: X.509

X.509 format (high level)

Version, Validity, Serial Number, and others..
CA Identity
Subject Identity
Subject Public Key
CA Siganture

X.509 certificate: real example

```
marlon@ubuntu:~/Desktop$ openssl x509 -in www.facebook.com
-----BEGIN CERTIFICATE-----
```

```
MIIGMjCCBRqgAwIBAgIQDG/IWVf6H1/JZyyf5lzb5jANBgkqhkiG9w0BAQUFADBm
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMSUwIwYDVQQDExxEaWdpQ2VydCBIaWdoIEFzc3VyYW5j
ZSBDQS0zMB4XDTEwMTExNTAwMDAwMFoXDTEzMTIwMjIzNTk1OVowajELMAkGA1UE
BhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExEjAQBgNVBAcTCVBhbG8gQWx0bzEX
MBUGA1UEChMORmFjZWJvb2ssIEluYy4xGTAXBgNVBAMTEHd3dy5mYWNlYm9vay5j
b20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAMHffWNBvcTk+mUzE3jVYjeW
p2HzsZa/I466h6ftB/neLeuox7ytd6ZejQMDNuNN99Dxq2byty4zFr4mD11BFn//
twCe+g6ZFWxSGtcKxq375AciP9sEpLZppe3Wh7aIxYP16Maz/8AOH52jhXDtonYU
e3A+77BCCzjWggAj3WN1AgMBAAGjggNaMIIDVjAfBgNVHSMEGDAWgBRQ6nOJ2yn7
EI+e5QEg1N55mUiD9zAdBgNVHQ4EFgQUqldKM7bs1W6BE6Y2XvR7Q1jzj0QwKQYD
VR0RBCIwIIIQd3d3LmZhY2Vib29rLmNvbYIMZmFjZWJvb2suY29tMHsGCCsGAQUF
BwEBBG8wbTAkBggrBgEFBQcwAYYYaHR0cDovL29jc3AuZGlnaWNlcnQuY29tMEUG
CCsGAQUFBzAChjlodHRwOi8vY2FjZXJ0cy5kaWdpY2VydC5jb20vRGlnaUNlcnRI
aWdoQXNzdXJhbmNlQ0EtMy5jcnQwDgYDVR0PAQH/BAQDAgWgMAwGA1UdEwEB/wQC
MAAwZQYDVR0fBF4wXDAsoCqgKIYmaHR0cDovL2NybDMuZGlnaWNlcnQuY29tL2Nh
My0yMDEwaS5jcmwwLKAqoCiGJmh0dHA6Ly9jcmw0LmRpZ2ljZXJ0LmNvbS9jYTMt
MjAxMGkuY3JsMIIBxgYDVR0gBIIBvTCCAbkwggG1BgtghkgBhv1sAQMAATCCAaQw
OgYIKwYBBQUHAgEWLmh0dHA6Ly93d3cuZGlnaWNlcnQuY29tL3NzbC1jcHMtcmVw
b3NpdG9yeS5odG0wggFkBggrBgEFBQcCAjCCAVYeggFSAEEAbgB5ACAAdQBzAGUA
IABvAGYAIAB0AGgAaQBzACAAQwBlAHIAdABpAGYAaQBjAGEAdABlACAAYwBvAG4A
cwB0AGkAdAB1AHQAZQBzACAAYQBjAGMAZQBwAHQAYQBuAGMAZQAgAG8AZgAgAHQA
aABlACAARABpAGcAaQBDAGUAcgB0ACAAQwBQAC8AQwBQAFMAIABhAG4AZAAgAHQA
aABlACAAUgBlAGwAeQBpAG4AZwAgAFAAYQByAHQAeQAgAEEAZwByAGUAZQBtAGUA
bgB0ACAAdwBoAGkAYwBoACAAbABpAG0AaQB0ACAAbABpAGEAYgBpAGwAaQB0AHkA
IABhAG4AZAAgAGEAcgBlACAAaQBuAGMAbwByAHAAbwByAGEAdABlAGQAIABoAGUA
cgBlAGkAbgAgAGIAeQAgAHIAZQBmAGUAcgBlAG4AYwBlAC4wHQYDVR0lBBYwFAYI
KwYBBQUHAwEGCCsGAQUFBwMCMA0GCSqGSIb3DQEBBQUAA4IBAQAlM16QP6OC/t6S
Op4S9+8Wao26ZqBarmZ2vEoSE+OS1vcPlLwBlSDo8P2sZt4kGK/uor9fo+xectYg
GtLGjwcNev+lj3OHfO6ZgQBqjYCnjcAAFsUd2AY39+wD6KKOQFyVdQwUAdF1p1aY
8DggH3cVeau14wQKd8nDtZlXdk8ObncaYTdvmrpTUT9RPpXAtMQgl+kmEODDGeRB
2Sb3OUvyoaTDtQXFvuJlhcspgGHW14e6yCX+hXG70mZjUkkLHWqAYkM8J/w8Khwu
gqeCEJjrS1oyfLGPXDkAxC9xtb3+v2DdAEOj8xCWg/hvleSrYh1SBXmU1zHyHHVE
yieOb6nD
```

```
-----END CERTIFICATE-----
```

Certificate Viewer: www.facebook.com

General | Details

This certificate has been verified for the following usages:

SSL Server Certificate

Issued To

Common Name (CN)	www.facebook.com
Organization (O)	Facebook, Inc.
Organizational Unit (OU)	<Not Part Of Certificate>
Serial Number	0C:6F:C8:59:57:FA:1F:5F:C9:67:2C:9F:E6:5C:DB:E6

Issued By

Common Name (CN)	DigiCert High Assurance CA-3
Organization (O)	DigiCert Inc
Organizational Unit (OU)	www.digicert.com

Validity Period

Issued On	11/15/10
Expires On	12/3/13

Fingerprints

SHA-256 Fingerprint	BB A9 12 B4 FE 2F 26 88 7D 79 0B C4 2F 7A 98 7B C8 D8 1C 21 B1 90 C4 46 5B C3 1A 2C 5B 6F D2 31
SHA-1 Fingerprint	63 08 84 E2 79 CB 11 07 F1 FB 8A 6B 11 A6 4D 1B 14 76 3F 8E

Close

X.509 certificate: real example

Version: 3 (0x2)
Serial Number:
  0c:6f:c8:59:57:fa:1f:5f:c9:67:2c:9f:e6:5c:db:e6
Signature Algorithm: sha1WithRSAEncryption
Issuer: C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert High Assurance CA-3
Validity
  Not Before: Nov 15 00:00:00 2010 GMT
  Not After : Dec 2 23:59:59 2013 GMT
Subject: C=US, ST=California, L=Palo Alto, O=Facebook, Inc., CN=www.facebook.com
Subject Public Key Info:
  Public Key Algorithm: rsaEncryption
  RSA Public Key: (1024 bit)
    Modulus (1024 bit):
      00:c1:df:7d:63:41:bd:c4:e4:fa:65:33:13:78:d5: **(... cut…)** 0b:38:d6:82:00:23:dd:63:75
    Exponent: 65537 (0x10001)

X509v3 extensions: **(cut)**
  X509v3 Subject Key Identifier:
    AA:57:4A:33:B6:EC:D5:6E:81:13:A6:36:5E:F4:7B:43:58:F3:8F:44
  X509v3 Subject Alternative Name:
    DNS:www.facebook.com, DNS:facebook.com
  X509v3 Key Usage: critical
    Digital Signature, Key Encipherment
  X509v3 Basic Constraints: critical
    CA:FALSE
  X509v3 Extended Key Usage:
    TLS Web Server Authentication, TLS Web Client Authentication

Signature Algorithm: sha1WithRSAEncryption
25:33:5e:90:3f:ad:02:fe:de:92:d2:9e:12:f7:ef:16:6a:8d: **(... cut…)** 8e:6f:a9:c3

Certificate Signing Request

- A certificate signing request (also CSR or certification request) is a message sent from an applicant to a certificate authority in order to apply for a digital identity certificate
- The most common format for CSRs is the PKCS#10 specification
- Operations:
  - the applicant first generates a key pair, keeping the private key secret
  - the applicant generates a CSR contains information identifying herself (X.509 subject field), optional X.509 extensions (e.g. key usage: RSA authentication for web servers) and the public key chosen by the applicant
  - The CSR may be accompanied by other credentials or proofs of identity required by the certificate authority, and the certificate authority may contact the applicant for further information

Let’s build our own certification authority

OPENSSL X509 TUTORIAL

OpenSSL

- OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them
  - www.openssl.org
- Main component
  - Cryptography library: `libcrypto`
  - SSL/TLS protocol library: `libssl`
  - `openssl` program
- The openssl program is a command line tool for using the various cryptography functions of OpenSSL's crypto library from the shell. It can be used for
  - Creation and management of private keys, public keys and parameters
  - Public key cryptographic operations
  - Creation of X.509 certificates, CSRs and CRLs
  - Calculation of Message Digests
  - Encryption and Decryption with Ciphers
  - SSL/TLS Client and Server Tests
  - Handling of S/MIME signed or encrypted mail
  - Time Stamp requests, generation and verification

Create a CA and sign certificate request with openssl

- Typical workflow
  1. Generate the RSA key pair for our CA
  2. Create a self-signed certificate for our CA
  3. Generate the RSA key pair for the web server
  4. Generate a CSR for the web server
  5. Sign the CSR with the CA private key

- Very simple Lab-pki
  - Create the CA and issue the certificates (single level certification ROOT_CA→certificate) with openssl from the host machine
  - Create a netikit lab (Lab9-pki) with just one VM (with a TAP 10.0.0.1,10.0.0.2) that will be our test web server
  - Setup Apache2 for a HTTPS website

Create the CA keys

Prepare our CA folder and the serial number file

```
marlon@marlon-vmxbn:~/Labs$ mkdir cgrlCA
marlon@marlon-vmxbn:~/Labs$ cd cgrlCA/
marlon@marlon-vmxbn:~/Labs/cgrlCA$ echo -e "01\n" > serial
```

Create the CA key pair

```
marlon@marlon-vmxbn:~/Labs/cgrlCA$ openssl genrsa -out ca.key 2048
Generating RSA private key, 2048 bit long modulus
..................................+++
........+++
e is 65537 (0x10001)
```

Generate the CA self signed certificate

This command will create a self signed certificate, i.e. a certificate where the issuer and the subject are the same entities

```
marlon@marlon-vmxbn:~/Labs/cgrlCA$ openssl req -new -x509 -days
3650 -key ca.key -out ca.crt
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]:IT
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:Rome
Organization Name (eg, company) [Internet Widgits Pty Ltd]:cgrlCA
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:cgrl-cert-authority
Email Address []:ca@cgrl.edu
```

Let’s take a look at our first certificate

```
marlon@marlon-vmxbn:~/Labs/cgrlCA$ openssl x509 -in ca.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            b6:ef:85:6f:71:e5:68:bb
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=IT, ST=Some-State, L=Rome, O=cgrlCA, CN=cgrl-cert-authority
                                                       emailAddress=ca@cgrl.edu
        Validity
            Not Before: May 24 10:44:00 2012 GMT
            Not After : May 22 10:44:00 2022 GMT
        Subject: C=IT, ST=Some-State, L=Rome, O=cgrlCA, CN=cgrl-cert-authority/
                                                       emailAddress=ca@cgrl.edu
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:a1:2c:f1:bf:a2:af:4a:3a:6e:f7:e7:13:b5:42:
                    32:4c:2c:d2:3b:0f:09:68:d6:67:6e:af:05:23:a8:
                    59:eb:ef:85:19:7c:75:18:   Cut!
```

Let's make the web server keys and CSR

Create the subject's (i.e. our web server) key pair

```
marlon@marlon-vmxbn:~/Labs/cgrlCA$ openssl genrsa -out server.key 1024
Generating RSA private key, 1024 bit long modulus
.++++++
.....................................................................++++++
e is 65537 (0x10001)
```

Create the subject's CSR. This certificate will be signed with the CA's private key

```
marlon@marlon-vmxbn:~/Labs/cgrlCA$ openssl req -new -key server.key -out
server.csr

Country Name (2 letter code) [AU]:IT
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:Rome
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:testssl.cgrl.edu
Email Address []:testssl@cgrl.edu
```

This has to be
The web site FQDN

CSR signing

This command will sign the CSR with the CA's private key

```
marlon@marlon-vmxbn:~/Labs/cgrlCA$ openssl x509 -req -in server.csr -out
server.crt -sha1 -CA ca.crt -CAkey ca.key -CAserial serial -days 3650
Signature ok
subject=/C=IT/ST=Some-State/L=Rome/O=Internet Widgits Pty Ltd/
CN=testssl.cgrl.edu/emailAddress=testssl@cgrl.edu
Getting CA Private Key
```

Dump the signed certificate

```
marlon@marlon-vmxbn:~/Labs/cgrlCA$ openssl x509 -in server.crt -text -noout
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 3 (0x3)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=IT, ST=Some-State, L=Rome, O=cgrlCA, CN=cgrl-cert-authority/
                                                       emailAddress=ca@cgrl.edu
        Validity
            Not Before: May 24 10:50:25 2012 GMT
            Not After : May 22 10:50:25 2022 GMT
        Subject: C=IT, ST=Some-State, L=Rome, O=Internet Widgits Pty Ltd,
                                 CN=testssl.cgrl.edu/emailAddress=testssl@cgrl.edu
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
```

How to protect our web server

HTTPS SERVER WITH APACHE2

Let's configure Apache2

We are going to create a virtual host for the website "testssl.cgrl.edu" in the netkit lab "Lab9-pki"

Configuration file, keys and certificate already in server:root/
Webserver media file and index.html in server:/var/www/testssl

Set-up everything properly before enabling the new site

- Configuration file testssl.cgrl.edu goes into /etc/apache2/site-available
- Keys and Certificate in the proper directory (see the conf file)

Run the following commands:

```
server# a2ensite testssl.cgrl.edu

server# a2enmod ssl

server# /etc/init.d/apache2 start
```

Enable our HTTPS web site

Enable Apache2 SSL module

Start Apache2
(or "restart" if already up)

testssl.cgrl.edu config file

```
IfModule mod_ssl.c>
<VirtualHost _default_:443>
DocumentRoot "/var/www/testssl"

ServerName testssl.cgrl.edu:443
ServerAdmin testssl@cgrl.edu

SSLEngine On
SSLCipherSuite HIGH:MEDIUM
SSLProtocol all -SSLv2
SSLCertificateFile /etc/apache2/ssl/server.crt
SSLCertificateKeyFile /etc/apache2/ssl/server.key
SSLCertificateChainFile /etc/apache2/ssl/ca.crt
SSLCACertificateFile /etc/apache2/ssl/ca.crt

<Directory "/var/www/testssl">
        Options Indexes
        AllowOverride None
        Allow from from all
        Order allow,den
</Directory>
</VirtualHost>
</IfModule>
```

Connect to the server

Unknown CA (of course…)

You can also manually and permanently add the certificate before trying to connect

Note: append the following line to the file `/etc/hosts` on the host machine

```
testssl.cgrl.edu 10.0.0.2
```

TLSv1 trace with our certificate

nk_tap_marlon [Wireshark 1.6.2]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: ssl Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000473	10.0.0.1	10.0.0.2	TLSv1	235	Client Hello
6	0.011022	10.0.0.2	10.0.0.1	TLSv1	1514	Server Hello
8	0.011170	10.0.0.2	10.0.0.1	TLSv1	844	Certificate, Server Key Exchange, Server Hello Done
10	0.013859	10.0.0.1	10.0.0.2	TLSv1	264	Client Key Exchange, Change Cipher Spec, Encrypted Handsha
11	0.019209	10.0.0.2	10.0.0.1	TLSv1	348	Encrypted Handshake Message, Change Cipher Spec, Encrypted
12	0.019530	10.0.0.1	10.0.0.2	TLSv1	439	Application Data
16	0.070438	10.0.0.2	10.0.0.1	TLSv1	678	Application Data, Application Data, Application Data, Appl
17	0.080485	10.0.0.1	10.0.0.2	TLSv1	455	Application Data

```
Certificates Length: 1750
▼ Certificates (1750 bytes)
    Certificate Length: 769
  ▶ Certificate (pkcs-9-at-emailAddress=testssl@cgrl.edu,id-at-commonName=testssl.cgrl.edu,id-at-organizatior
    Certificate Length: 975
  ▶ Certificate (pkcs-9-at-emailAddress=ca@cgrl.edu,id-at-commonName=cgrl-cert-authority,id-at-organizationNa
▶ TLSv1 Record Layer: Handshake Protocol: Server Key Exchange
▶ TLSv1 Record Layer: Handshake Protocol: Server Hello Done
```

issuer

subject

```
0000  16 03 01 06 dd 0b 00 06  d9 00 06 d6 00 03 01 30   ........ .......0
0010  82 02 fd 30 82 01 e5 02  01 03 30 0d 06 09 2a 86   ...0.... ..0...*.
0020  48 86 f7 0d 01 01 05 05  00 30 7c 31 0b 30 09 06   H....... .0|1.0..
```

Frame (844 bytes) Reassembled TCP (2173 bytes)

Handshake protocol message (ssl.han... Packets: 222 Displayed: 41 Marked: 0 Dropped: 0 Profile: Default

HTTP plaintext auth over SSL

- Safest way to authenticate via HTTP, better then digest auth
- You first create a secure channel with the authenticated web server
- You send authentication credential in clear (from the HTTP point of view) but inside the secure (encrypted/authenticated) channel
- The test website already have the following password protected directory

```
<Directory "/var/www/testssl/secret">
    AuthType Basic
    AuthName "Username and Password Required"
    AuthUserFile /etc/apache2/.htpasswd
    Require valid-user
</Directory>
```

To try it you need to grant access to a new user, for example: uid “007” password “jamesbond”

```
server# htpasswd -c -m /etc/httpd/.htpasswd 007
New password:
```

Client authentication via X509 certificate

- The client may authenticate itself with a X509 certificate
- To do so we need to
  1. Configure the web server to force SSL client authentication

```
<Directory "/var/www/testssl/cert-required">
    SSLVerifyClient require
    SSLVerifyDepth 1
</Directory>
```

  2. Create a client certificate and configure the web browser to use it (exported it in PCKS 12 format. NOTE: to use it with firefox you need to enable SSL renegotiation. With (my) chrome (v. 15.0.874.106 (Developer Build 107270 Linux) Ubuntu 11.10) it's already OK)

```
server# openssl genrsa -out client.key 1024
server# openssl req -new -key client.key -out client.csr
server# openssl x509 -req -in client.csr -out client.crt -sha1 -CA
ca.crt -CAkey ca.key -CAserial serial -days 3650
server# openssl pkcs12 -export -in client.crt -inkey client.key -
out client.p12
```