

CT437  
COMPUTER SECURITY AND FORENSIC COMPUTING

DIGITAL CERTIFICATES

Dr. Michael Schukat

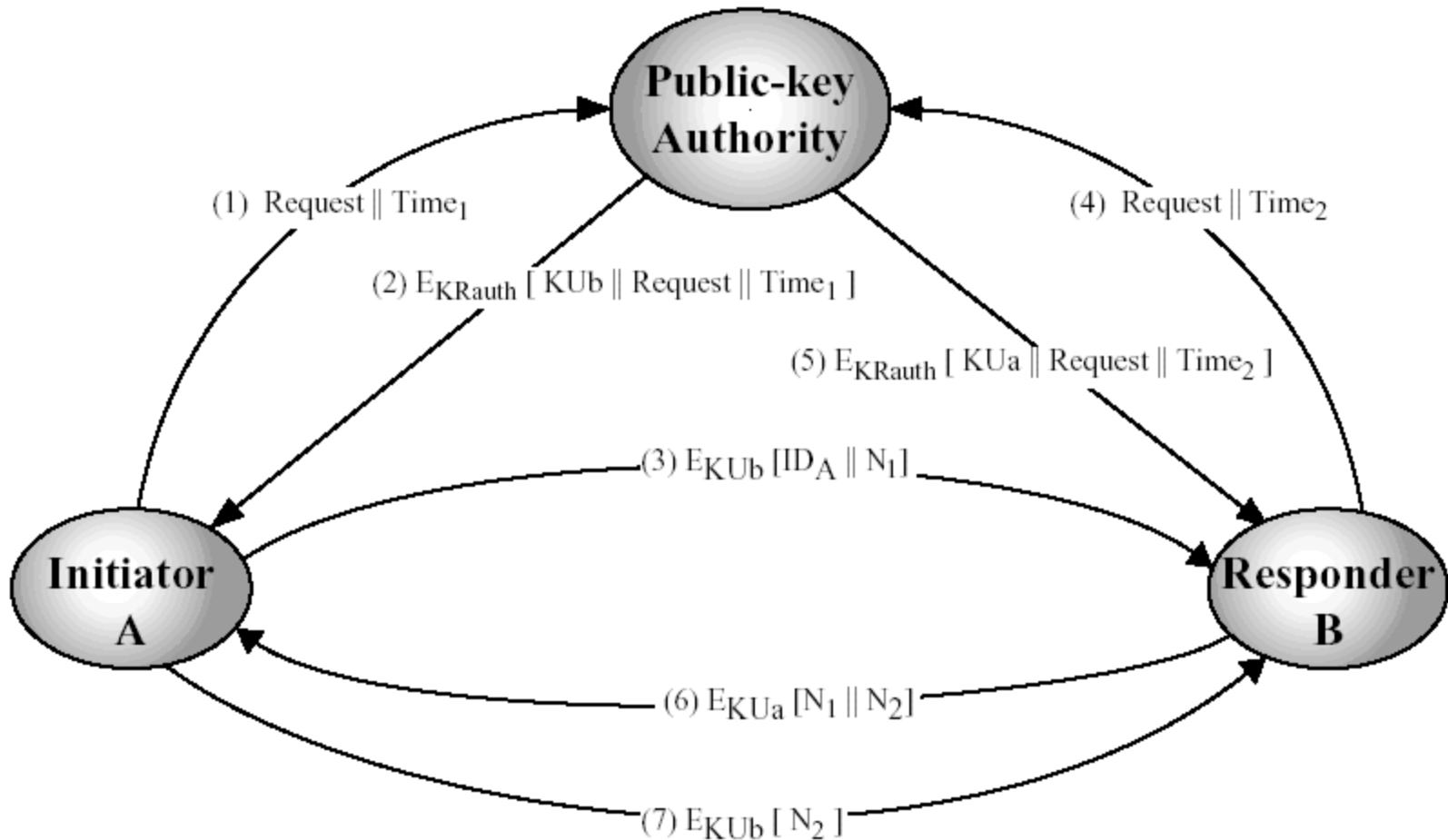


# Lecture Content

2

- Recap motivation digital certificates
- Digital certificates and certificate authorities
  - ▣ Concepts
  - ▣ Applications
  - ▣ Case studies

# Recap: Key Management via Public-Key Authority



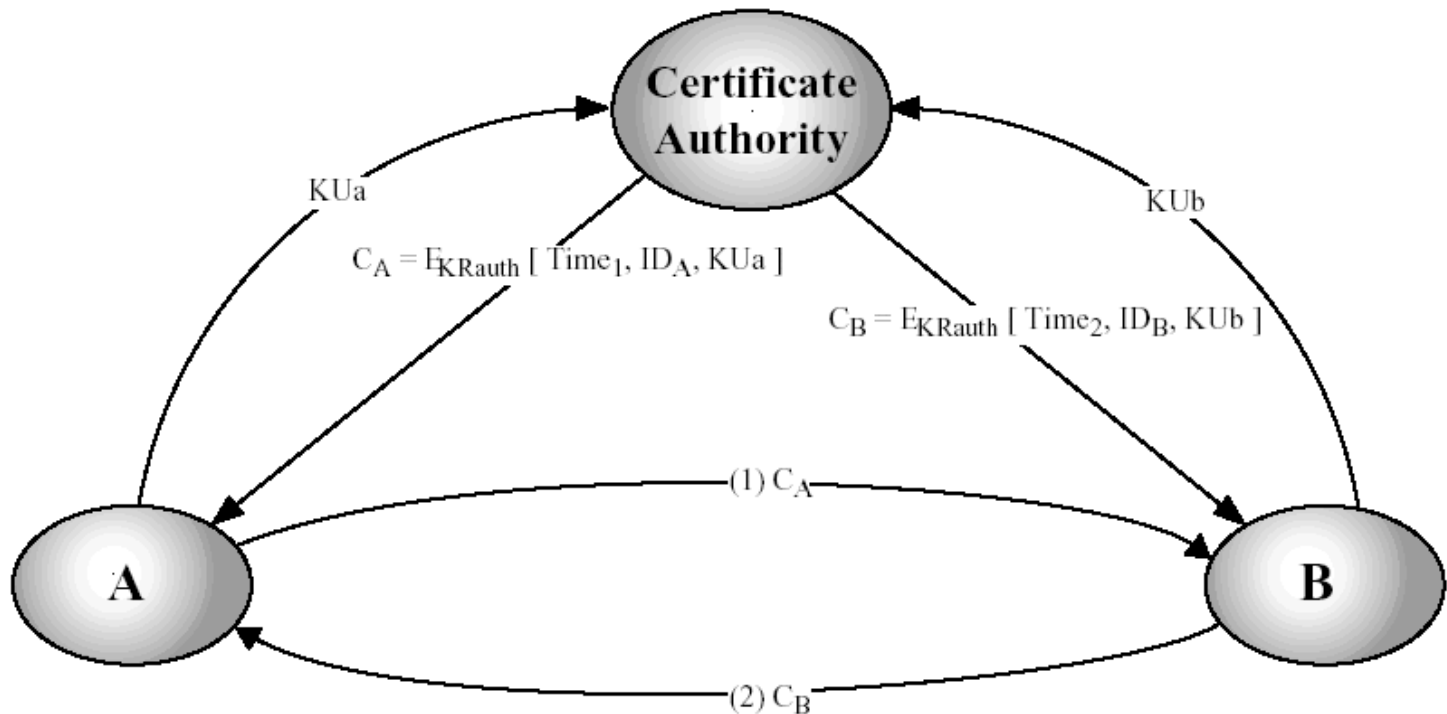
□ Please see also lecture notes “Public Key Encryption”

# Recap: Key Management via Public-Key Authority

- Drawback of public-key authority:  
**Authority is a bottleneck! If it is compromised (e.g. via a DoS attack), public keys cannot be requested or distributed**
- Therefore: Introduction of certificates, that can be used by participants to exchange keys without contacting a public-key authority
- Requirements:
  - ▣ Any participant can read a certificate to determine the name and public key of the certificate's owner
  - ▣ Any participant can verify that the certificate originated from the certificate authority and is not counterfeit
  - ▣ Only the certificate authority can create, renew and revoke certificates
  - ▣ Any participant can verify the validity (i.e., expiration or revocation) of the certificate

# Recap: Key Management via Certificate Authority

- Architecture allows exchange of public-key certificates (PKC):



# Recap: Example for a Simple XML-Based Signature: Plaintext

```
<SimpleSignature>
  <Authority> NUI-Galway </Authority>
  <SignatureType> SimpleSignature </SignatureType>
  <Created> 15-NOV-2019 </Created>
  <Expires> 14-NOV-2020</Expires>
  <OwnerName> William Simpson </OwnerName>
  <KeyType> RSA </KeyType>
  <KeyLength> 256 </KeyLength>
    <PublicKey>
      gHJgjh57JKf#j'\;gkwg@45tRET46$Ed
    </PublicKey>
</SimpleSignature>
```

# Recap: Example for a Simple XML-Based Signature: Ciphertext

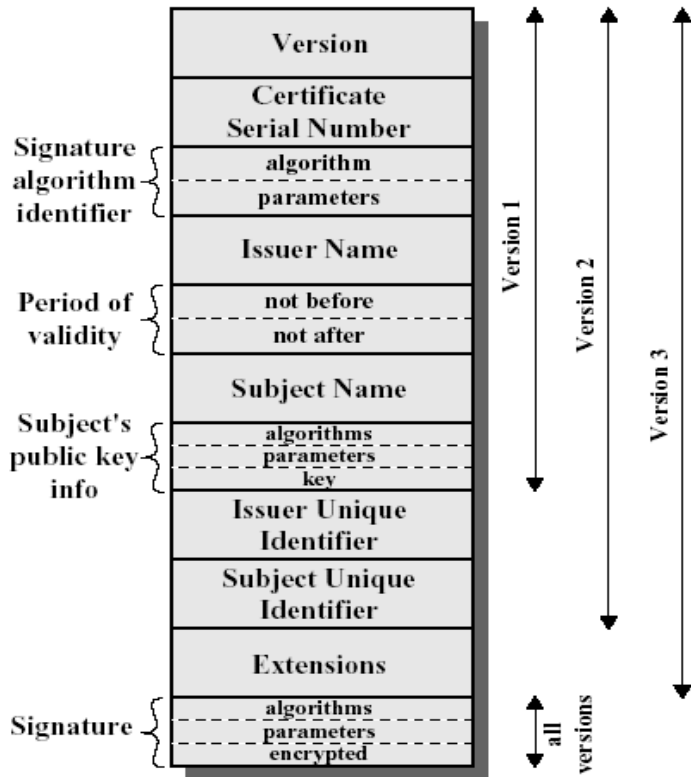
hi6IGHJ^gu#"':HGLFdyUf56EEdx3X5XxXuAzyI;\*6/.,:g  
wqui^09udfsqfhaspfaj#w994HK51'fjg095u321\er3f2875  
gyor23ro32rj6yhggIGUoowqru07t99Y)\*-36wrqwUluill  
No891 u[ `[c0 t6Rt\*(v858e3w70-v794x3xz7c8c9799999s  
9udfsqfhaspfaj7t99 -v794x3xz7c8c9799 09udfsqfhaspfaj#  
w994HK51'fjg095u32nfjewYU87Deffe7s°%Rk936-J0D9d

# X.509 Certificates

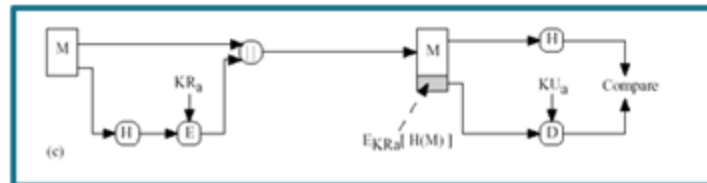
- X.509 is an International Telecommunication Union (ITU) standard defining the format of Public Key Certificates (PKC)
  - ▣ Public key management generally involves the use of PKCs
  - ▣ PKCs bind an identity (the subject) to a public key,
    - usually with other info such as period of validity, rights of use etc.
    - with all contents signed by a trusted Certification Authority (CA), the issuer
  - ▣ Therefore, X.509 certificates are also called **identity certificates**
  - ▣ In all PKC use cases (e.g., peer-to-peer data communication), involved parties either already know, or can securely obtain and verify the public key of the CA to verify the certificate
  
- X.509 certificates are widely used in secure email (S/MIME - Secure Multipurpose Internet Mail Extensions), secure web browsing (TLS / HTTPS), secure software patching, etc.



# X.509 Certificate Structure



- The certificate is issued by a CA, who signs the certificate
  - ▣ The certificate is hashed, and the hash is encoded (signed) by the CA using its private key
  - ▣ In the diagram below,  $M$  is the entire certificate excluding the signature, which in turn is the encrypted hash
- The certificate can be validated by anyone who has a trusted (!) copy of the issuer's (CA's) public key:

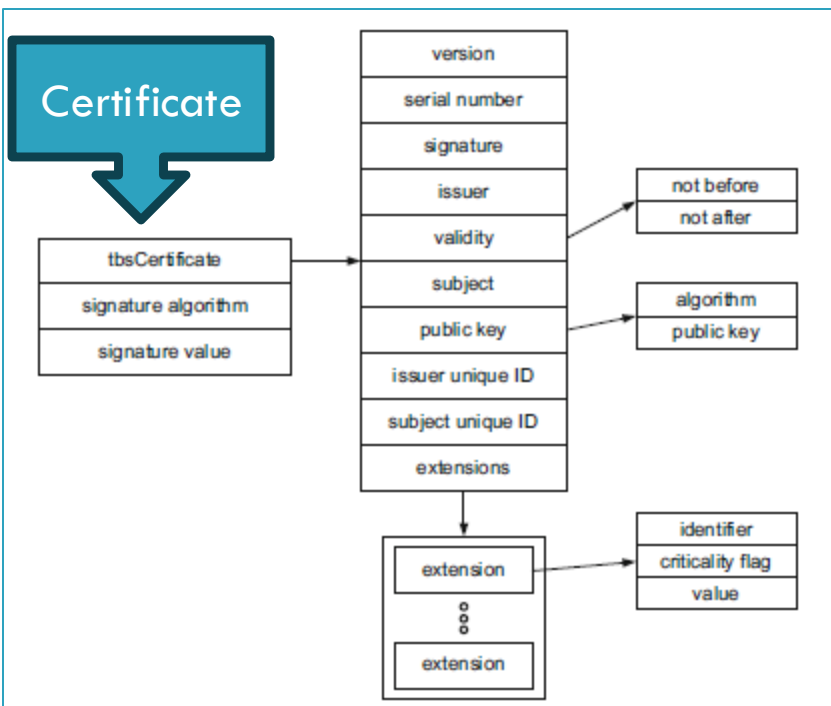


$KR_a$  = CA private key  
 $KU_a$  = CA public key

# X.509 Certificate Specification

10

- Digital certificates are described via ASN.1
- Abstract Syntax Notation One (ASN.1) is a standard interface description language for defining data structures that can be serialised and de-serialised in a cross-platform way (→ later)

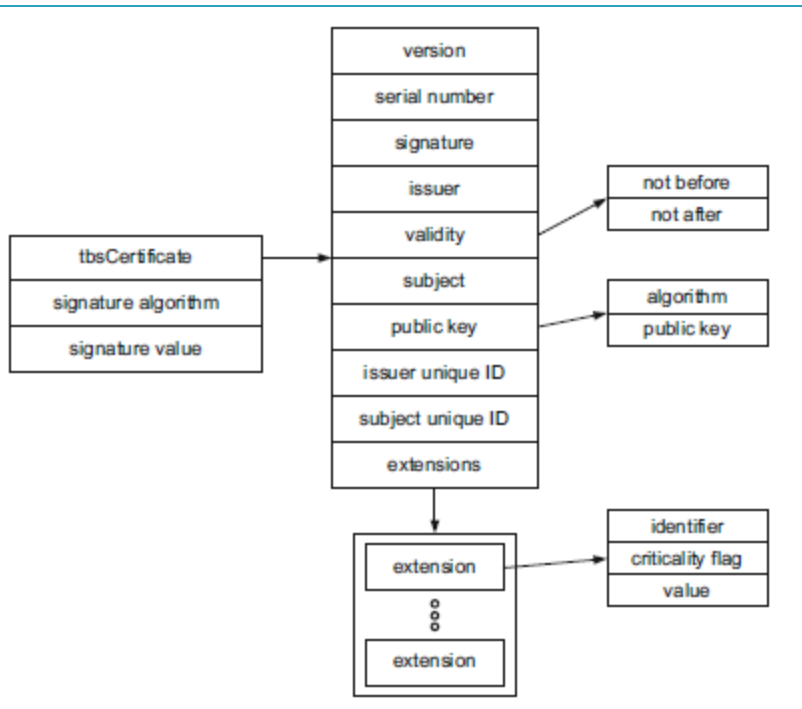


```
Certificate ::= SEQUENCE {  
    tbsCertificate      TBSCertificate,  
    signatureAlgorithm  AlgorithmIdentifier,  
    signatureValue      BIT STRING }
```

```
TBSCertificate ::= SEQUENCE {  
    version             [0] EXPLICIT Version DEFAULT v1,  
    serialNumber        CertificateSerialNumber,  
    signature            AlgorithmIdentifier,  
    issuer              Name,  
    validity            Validity,  
    subject             Name,  
    subjectPublicKeyInfo SubjectPublicKeyInfo,  
    issuerUniqueID      [1] IMPLICIT UniqueIdentifier OPTIONAL,  
    subjectUniqueID     [2] IMPLICIT UniqueIdentifier OPTIONAL,  
    extensions          [3] EXPLICIT Extensions OPTIONAL }
```

# X.509 Certificates and OID

11



- X.509 digital certificates contain various fields containing mandatory and optional attributes
  - ▣ Mainly extension are optional
- These attributes are described / encoded using **Object Identifiers (OID)** → next slide
- A digital certificate is a structured list of OIDs and attribute values
- This list is converted into a data structure encoded using BER (Basic Encoding Rules) → later

# Object Identifiers (OID)

- ❑ OIDs are a standardised identifier mechanism for naming any object, concept, or "thing" with a globally unambiguous persistent name
- ❑ OIDs are dotted numbers, with similar concepts often having identical or similar OID pre-fixes
- ❑ X.509 attribute values are either instances of primitive data types (e.g., an integer for version number), or are described by an OID
- ❑ For example, all (standardised) cryptographic algorithms used / supported by X.509 have their unique OID – see also the table above

Algorithm	Type	OID
MD5	Cryptographic hash function	1.2.840.113549.2.5
SHA1	Cryptographic hash function	1.3.14.3.2.26
SHA256	Cryptographic hash function	2.16.840.1.101.3.4.2.1
SHA384	Cryptographic hash function	2.16.840.1.101.3.4.2.2
SHA512	Cryptographic hash function	2.16.840.1.101.3.4.2.3
SHA256withDSA	Digital signature	2.16.840.1.101.3.4.3.2
SHA256withECDSA	Digital signature	1.2.840.10045.4.3.2
SHA384withECDSA	Digital signature	1.2.840.10045.4.3.3
SHA512withECDSA	Digital signature	1.2.840.10045.4.3.4
MD5withRSA	Digital signature	1.2.840.113549.1.1.4
SHA1withRSA	Digital signature	1.2.840.113549.1.1.5
SHA1withDSA	Digital signature	1.2.840.10040.4.3
SHA1withECDSA	Digital signature	1.2.840.10045.4.1
AES with 128 bit key in ECB mode	Secret key encryption	2.16.840.1.101.3.4.1.1
AES with 256 bit key in CBC mode	Secret key encryption	2.16.840.1.101.3.4.1.42
HMAC-MD5	MAC	1.3.6.1.5.5.8.1.1
HMAC-SHA1	MAC	1.3.6.1.5.5.8.1.2
RSA	Public key encryption	1.2.840.113549.1.1.1

# OIDs in Digital Certificates

- In the mock-up example attribute OIDs are replaced with their **name**
- **Other descriptors** don't appear in a certificate and are only added to increase readability
- Note that **Issuer** / **Subject** and **NotBefore** / **NotAfter** attributes can be only distinguished via their position in the cert (i.e, **Subject** appears after the **Issuer**; **notAfter** appears after **NotBefore**)

**Version:** 3

**Serial Number:** 3c:50:33:c2:f8:e7:5c:ca:07:c2:4e3:f2:e8:0e:4f

**Issuer:** O=VeriSign, Inc., OU=VeriSign Trust Network,

OU=www.verisign.com

CN=VeriSign Class 1 CA

**Validity NotBefore:** Jan 13 00:00:00 2021 GMT **NotAfter:** Mar 13 23:59:59 2026 GMT

**Subject:** O=VeriSign, Inc., OU=VeriSign Trust Network, OU=www.verisign.com CN=Lawrie Brown

Email=lawrie.brown@canb.aug.org.au

**Subject Public Key Info:** rsaEncryption RSA Public Key: (512 bit):

00:98:f2:89:c4:48:e1:3b:2c:c5:d1:48:67:80:53: d8:eb:4d:4f:ac:31:a9:fd:

11:68:94:ba:44:d8:48: 46:0d:fc:5c:6d:89:47:3f:9f:d0:c0:6d:3e:9a:8e:ec:

82:21:48:9b:b9:78:cf:aa:09:61:92:f6:d1:cf: 45:ca:ea:8f:df

**Signature Algorithm:** SHA1 withRSA

**Signature Value:** 5a:71:77:c2:ce:82 ...



OID of Version:  
2.5.29.19

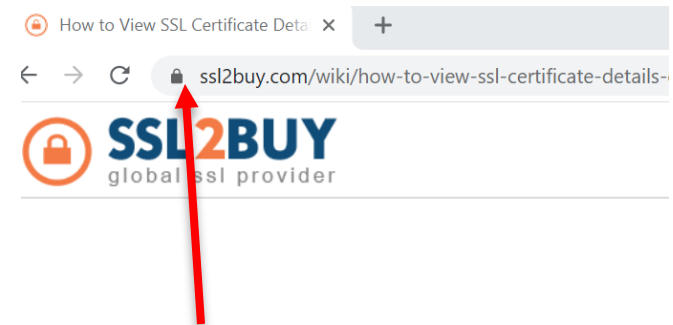
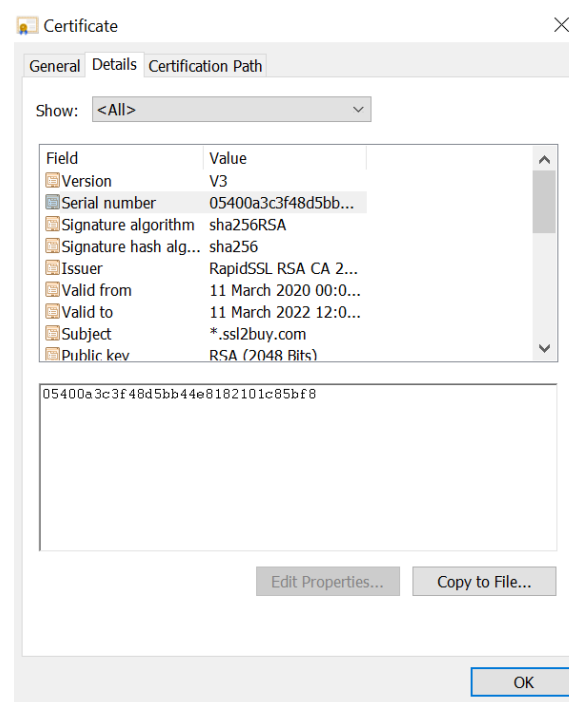
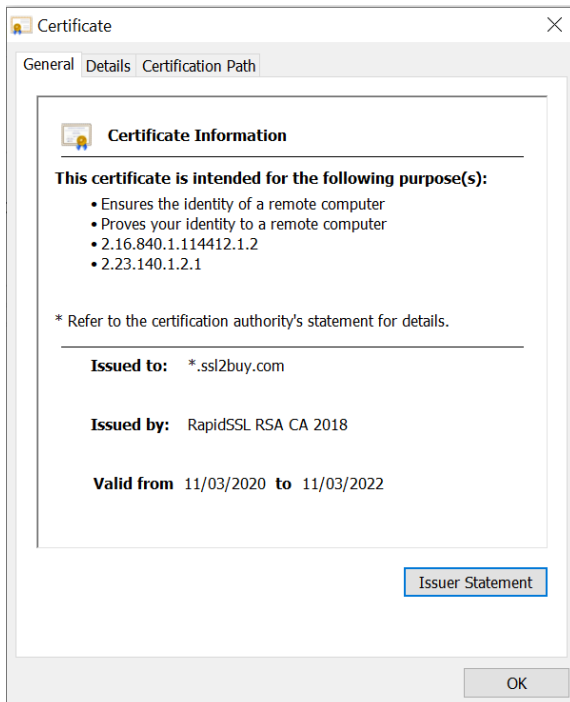
# In Class Activity: Inspect Digital Certificates on your Device / Browser

14

- ❑ Android (version 11):
  - ❑ Open Settings
  - ❑ Tap “Security”
  - ❑ Tap “Encryption & credentials”
  - ❑ Tap “Trusted credentials.” This will display a list of all trusted certs on the device
- ❑ In Chrome (Windows OS):
  - ❑ Goto Settings
  - ❑ Open “Security and Privacy” and “Security”
  - ❑ Open “Manage device certificates”
  - ❑ iOS devices require you to open the keystore
- ❑ iOS devices:
  - ❑ Tap Settings > General > About
  - ❑ Scroll to the bottom of the list
  - ❑ Tap Certificate Trust Settings
  - ❑ Follow the link
- ❑ Generic: <https://www.ssllabs.com/ssltest/?form=MG0AV3>

# Example: X.509 Certificates in Web Browsers

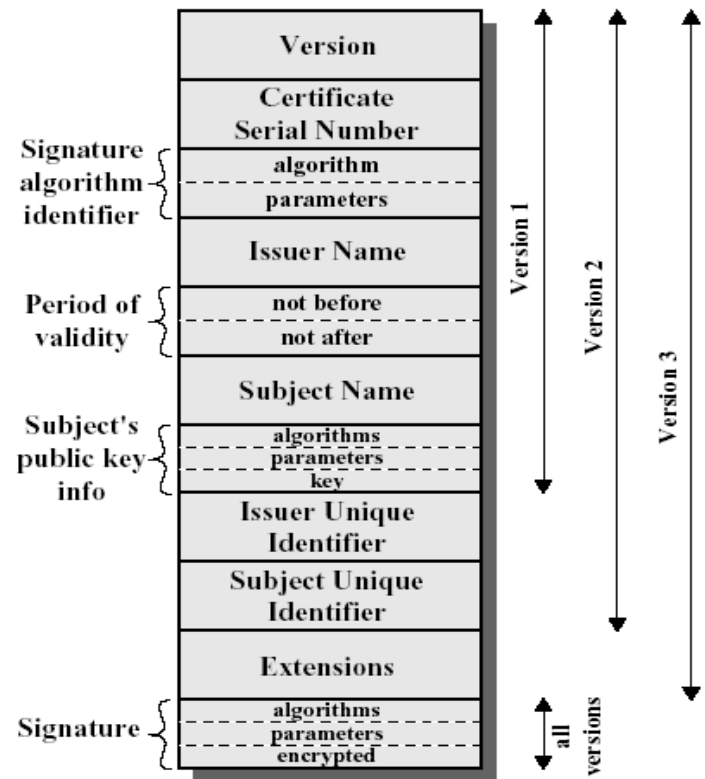
- In Chrome: see <https://www.ssl2buy.com/wiki/how-to-view-ssl-certificate-details-on-chrome-56>



# X.509 Certificates in Detail: Field *version*

16

- X.509 certificates went through three iterations before v3 was finally released in 1996 (!)
- The value of the *version* field (OID 2.5.29.19) is an integer
  - ▣ X.509v1 → 0
  - ▣ X.509v2 → 1
  - ▣ X.509v3 → 2





# Fields *Issuer* and *Subject*

17

- **Issuer** is the certificate authority (CA) that signed the certificate

- **Subject** is the owner of the cert

- Both their descriptions are provided via a string called the **Distinguished Name (DN)**

- A DN is a sequence of OID encoded attributes and their values

- Example: CN=Alice, OU=Administration, O=TU Darmstadt, C=DE

- ▣ This DN describes a person with common name (CN) Alice, who belongs to the organisational unit (OU) “administration” of the organization (O) “TU Darmstadt” that operates in the country (C) Germany

- ▣ Here the DN reflects a logical hierarchy of a person belonging to an organisational unit which is part of an organisation located in a country

- ▣ The DN as string would look like “2.5.4.3Alice2.5.4.11Administration ...”

Attribute type	String representation	OID
countryName	C	2.5.4.6
organizationName	O	2.5.4.10
organizationalUnitName	OU	2.5.4.11
commonName	CN	2.5.4.3
localityName	L	2.5.4.7
stateOrProvinceName	ST	2.5.4.8

# Field *serialNumber*

18

- The certificate issuer assigns a unique serial number to each signed certificate, composed as follows:
  - ▣ *SerialNumber* (OID 2.5.4.5)
  - ▣ a positive 20 byte long integer
  - ▣ E.g. “2.5.4.501234567890123456789”
    - As we will see later, each item is in fact represented as a Type-Length-Value triplet
- The serial number field is mandatory
- Therefore, the combination of the issuer name **and** the serial number uniquely identifies a certificate
  - ▣ Consider a subject could have multiple certificates signed by the same CA
  - ▣ Note that different CA can issue a certificate that has the same serial number

# Field *signature*

19

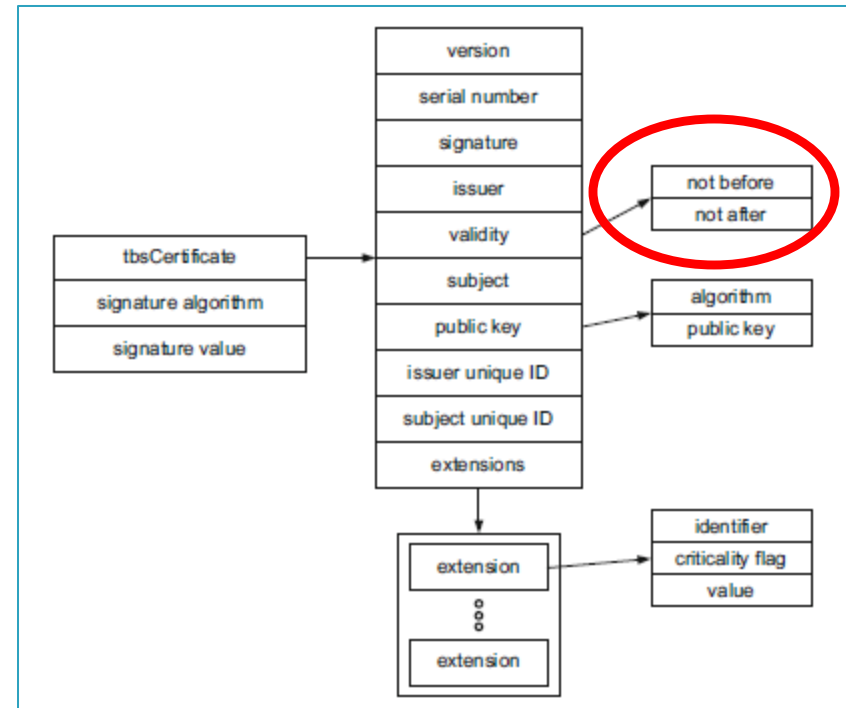
- The issuer of an X.509 certificate signs the certificate
- The mandatory field *signature* describes the signature algorithm that was used by the issuer to sign the certificate
- The field is of type *AlgorithmIdentifier* (OID 1.3.6.1.1.15.7)
- It is complemented by the OID of the signature algorithm that is used (see table) and optional additional parameters

Algorithm	Type	OID
MD5	Cryptographic hash function	1.2.840.113549.2.5
SHA1	Cryptographic hash function	1.3.14.3.2.26
SHA256	Cryptographic hash function	2.16.840.1.101.3.4.2.1
SHA384	Cryptographic hash function	2.16.840.1.101.3.4.2.2
SHA512	Cryptographic hash function	2.16.840.1.101.3.4.2.3
SHA256withDSA	Digital signature	2.16.840.1.101.3.4.3.2
SHA256withECDSA	Digital signature	1.2.840.10045.4.3.2
SHA384withECDSA	Digital signature	1.2.840.10045.4.3.3
SHA512withECDSA	Digital signature	1.2.840.10045.4.3.4
MD5withRSA	Digital signature	1.2.840.113549.1.1.4
SHA1withRSA	Digital signature	1.2.840.113549.1.1.5
SHA1withDSA	Digital signature	1.2.840.10040.4.3
SHA1withECDSA	Digital signature	1.2.840.10045.4.1
AES with 128 bit key in ECB mode	Secret key encryption	2.16.840.1.101.3.4.1.1
AES with 256 bit key in CBC mode	Secret key encryption	2.16.840.1.101.3.4.1.42
HMAC-MD5	MAC	1.3.6.1.5.5.8.1.1
HMAC-SHA1	MAC	1.3.6.1.5.5.8.1.2
RSA	Public key encryption	1.2.840.113549.1.1.1

# Field validity

20

- The validity field (OID 2.5.29.16) indicates the validity period of the certificate
- This field contains just two dates, which have no OID and are just referenced as *notBefore* and *notAfter*
- Between these two dates the certificate is valid unless it has been revoked (→ later)



# Field *subjectPublicKeyInfo*

21

- The *subjectPublicKeyInfo* field (OID 1.2.840.1.13549.1.1.1) contains the public key data that is certified by the certificate
- This data is described as a sequence containing the OID of an algorithm followed by optional parameters and the public key
- The example below shows the ASN.1 structure of an EC public key and its parameters

```
ECParameters ::= SEQUENCE {  
    version      ECPVer,  
    fieldID      FieldID,  
    curve        Curve,  
    base         ECPoint,  
    order        INTEGER,  
    cofactor    INTEGER OPTIONAL }
```

# Fields *issuerUniqueID* and *subjectUniqueID*

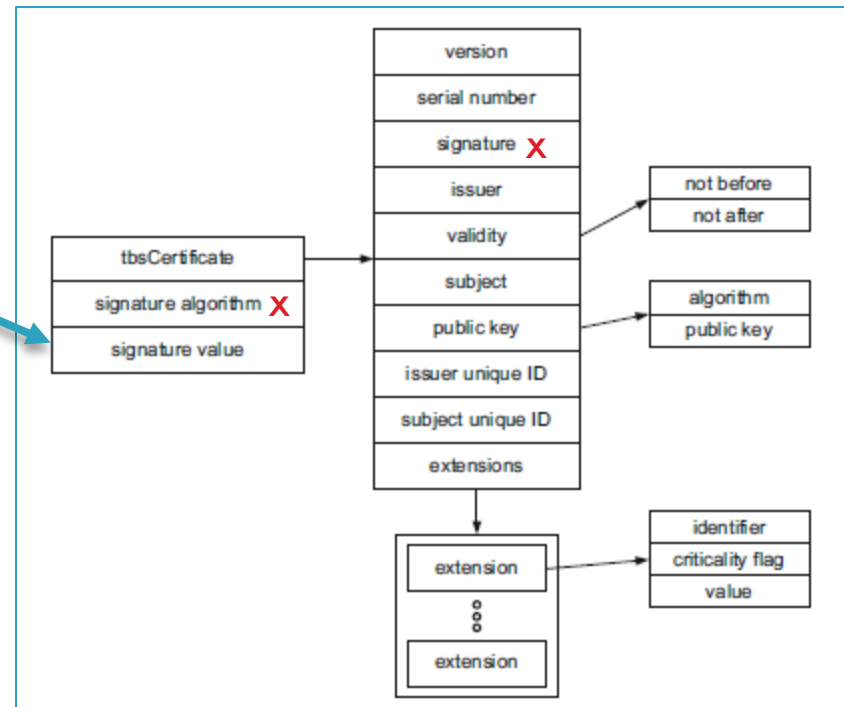
22

- The *subjectUniqueID* and *issuerUniqueID* fields were introduced with X.509v2
- It may happen that the same distinguished name is assigned to different entities
- For example, if a *subjectDN* is used twice by an issuer, then the owner of the corresponding certificate is not uniquely determined by the subject DN
- To make the owner description unique, the *subjectUniqueID* field may be added
- The content of that field is a binary string that is a unique identifier for the owner of the certificate
- Likewise, several issuers may share the same DN
- In this case the *issuerUniqueID* field resolves the situation
- **However, the use of these fields is not recommended because they make certificate use more complicated**

# Field *signatureAlgorithm* and *signatureValue*

23

- The signature algorithm that was used to sign the certificate is specified twice in an X.509 certificate:
  - In the *tbsCertificate* structure (under *signature*), as seen before
  - In the *signatureAlgorithm* field
- *signatureValue* holds the signature on the *tbsCertificate* content of the certificate, i.e. the encrypted hash of *tbsCertificate* (but not *signatureAlgorithm*)



# X.509 Certificate Extensions

24

- The contents of X.509 version 1 and version 2 certificates turned out to be insufficient in practice
- X.509v3 certificates may contain extensions which support various PKI processes
- The ASN.1 structure of X.509 certificate extensions can be seen below:
  - ▣ The first field in such an extension is *extnID*, which contains the OID of the extension
  - ▣ Next, any extension contains a criticality indicator *critical*
    - If its value is true, then all applications that use this certificate must evaluate the extension; If an application is unable to do so, then it must consider the certificate to be invalid
  - ▣ The third field contains the extension description

```
Extension ::= SEQUENCE {  
    extnID      OBJECT IDENTIFIER,  
    critical    BOOLEAN DEFAULT FALSE,  
    extnValue   OCTET STRING }
```



# Extension Field *AuthorityKeyIdentifier*

25

- Problem:
  - ▣ An issuer / CA may have multiple key pairs to sign a digital certificate
  - ▣ If a given certificate is to be validated, the correct public key must be chosen
  - ▣ The information in the issuer field just points to the CA, but not to the correct key
- Solution:
  - ▣ This extension, also known as AKI extension or AKIE, is to support applications in identifying the public key of the issuer, to be used to verify the certificate signature
- The authority key identifier extension must be present in any X.509v3 certificate unless the certificate is self-signed (→ later)
- Also, this extension must not be marked critical
- Typically, this value is a 20-byte SHA-1 hash of the public key belonging to the private key of the issuer that was used to sign the certificate
- Similarly, the extension field *SubjectKeyIdentifier* can be used to hash the subject's public key (more later)

# Extension Field *KeyUsage*

26

- The *KeyUsage* extension indicates what the public key contained in a certificate can be used for
- Possible uses are:
  - ▣ *digitalSignature*  
The public key can be used to verify digital signatures, for example, to validate the authenticity and origin of signed emails
  - ▣ *nonRepudiation*  
The public key can be used to verify signatures to provide nonrepudiation
    - E.g. denial of a digitally contract being signed
  - ▣ *keyEncipherment*  
The public key may be used to encrypt symmetric session keys
  - ▣ *dataEncipherment*  
The public key may be used to encrypt data
  - ▣ *keyAgreement*  
The public key may be used in a key agreement scheme (i.e., Diffie-Hellman)

```
KeyUsage ::= BIT STRING {  
    digitalSignature      (0),  
    nonRepudiation       (1),  
    keyEncipherment      (2),  
    dataEncipherment     (3),  
    keyAgreement         (4),  
    keyCertSign          (5),  
    cRLSign              (6),  
    encipherOnly         (7),  
    decipherOnly         (8) }
```

# Extension Field *KeyUsage*

27

## □ Possible uses are:

### ▣ keyCertSign

The private key corresponding to the public key in the certificate may be used to sign certificates. The public key is then used to verify certificate signatures

### ▣ cRLSign

The private key corresponding to the public key in the certificate may be used to sign certificate revocation lists (→ later)

### ▣ encipherOnly

Undefined in the absence of the keyAgreement bit

When the encipherOnly bit is asserted and the keyAgreement bit is also set, the subject public key may be used only for enciphering data while performing key agreement

### ▣ decipherOnly

ditto

## □ Many clients and applications evaluate the key usage extension

### ▣ Example: An email client that has access to several certificates of the recipient of an email can tell by the key usage extension which certificate is to be used for

- email encryption
- verifying signatures of received emails

```
KeyUsage ::= BIT STRING {  
    digitalSignature      (0),  
    nonRepudiation       (1),  
    keyEncipherment      (2),  
    dataEncipherment     (3),  
    keyAgreement         (4),  
    keyCertSign          (5),  
    cRLSign              (6),  
    encipherOnly         (7),  
    decipherOnly         (8) }
```

# Extension Field *SubjectAlternativeName*

28

- Up to now a subject is identified via its subject field that contains the distinguished name (DN) with all the aforementioned attributes
- This extension binds additional names to the public key in the certificate not covered by the DN
- Typical names contained in this extension are owner's

- email address
- IP address
- domain name (DNS names)
- uniform resource identifier (URIs)

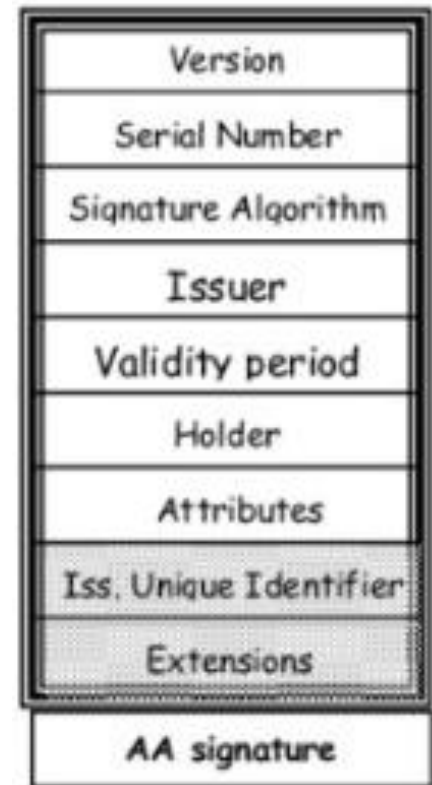
```
X509v3 Subject Alternative Name:  
DNS:*.wikipedia.org, DNS:*.m.mediawiki.org, DNS:*.m.wikibooks.org, DNS:*.m.wikidata.org,  
DNS:*.m.wikimedia.org, DNS:*.m.wikimediafoundation.org, DNS:*.m.wikinews.org, DNS:*.m.wikipedia.org,  
DNS:*.m.wikiquote.org, DNS:*.m.wikisource.org, DNS:*.m.wikiversity.org, DNS:*.m.wikivoyage.org, DNS:*.m.wiktionary.org,  
DNS:*.mediawiki.org, DNS:*.planet.wikimedia.org, DNS:*.wikibooks.org, DNS:*.wikidata.org, DNS:*.wikimedia.org,  
DNS:*.wikimediafoundation.org, DNS:*.wikinews.org, DNS:*.wikiquote.org, DNS:*.wikisource.org, DNS:*.wikiversity.org,  
DNS:*.wikivoyage.org, DNS:*.wiktionary.org, DNS:*.wmfusercontent.org, DNS:*.zero.wikipedia.org, DNS:mediawiki.org,  
DNS:w.wiki, DNS:wikibooks.org, DNS:wikidata.org, DNS:wikimedia.org, DNS:wikimediafoundation.org, DNS:wikinews.org,  
DNS:wikiquote.org, DNS:wikisource.org, DNS:wikiversity.org, DNS:wikivoyage.org, DNS:wiktionary.org,
```

- For example, if the public key in the certificate is used for authentication of the web server of an organisation, the DNS name or the IP address of that server is typically contained in this extension
  - Clients that connect securely to such a server verify that the IP address or the DNS name of the server matches the IP address or DNS name contained in this extension (more later)
- Example: UoG certificate

# Attribute Certificates

30

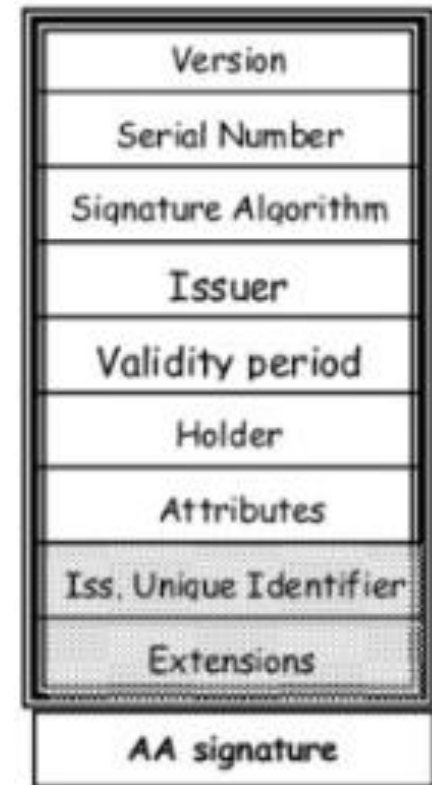
- An attribute certificate binds certain privileges or attributes to their owners
- It is signed by an attribute authority (AA)
- For example, attribute certificates are used in smartphones to provide apps with the permission to access certain phone resources, e.g., a user's address book
- In contrast to identity certificates, an attribute certificate does not contain the owner's public key
- On the other hand, identity certificates could be complemented by additional attributes encoded as new extension fields, and to some extent mimick attribute certificates
  - ▣ Such a certificate is also called a **combined certificate**



# Attribute Certificates

31

- Attributes are TLV triples as well, uniquely identified by their OID
- Attribute certificates are often used in conjunction with X.509 public key certificates
- For example, consider a firmware update for a mobile phone:
  - ▣ It is signed by its issuer and the signature verification key is authenticated by a certificate
  - ▣ In addition, an attached verifiable attribute certificate specifies whether or not this update may be used for a certain type of mobile phone



# Example Home Automation

32

- Consider a range of wireless IoT home automation devices that require
  1. secure inter-device communication
  2. end-point authentication
  3. optimised inter-device communication (i.e. the smart fridge and the electricity smart meter only exchange energy consumption data)
  4. the exclusion of 3<sup>rd</sup> party devices
- All devices are integrated in a home-automation network (HAN) and form P2P connections via some handshake protocol
- Each device has its own X.509 public key certificate
  - ▣ Certificates are exchanged between paired devices to provide end-point authentication (1) and secure session keys for secure wireless data communication (2)



# Example Home Automation

33

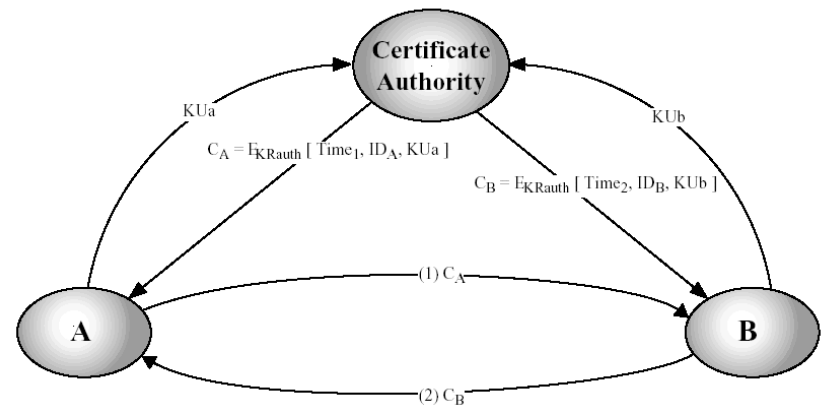
- However, in order to address 3. and 4., additional information must be encoded:
  - ▣ The device manufacturer
  - ▣ The device type
  - ▣ Rules that describe other devices it can talk to
- This info can be encoded in
  - ▣ an additional attribute certificate, or
  - ▣ additional extension fields of the public key certificate (creating a combined certificate)
- Subsequently, a device that during the handshake
  - ▣ cannot present these credentials, or
  - ▣ has the incorrect attribute values (e.g. different manufacturer)cannot complete the process and is excluded from the HAN



# Trust Models and Digital Certificates

34

- Problem: Public key cryptography (and subsequently digital certificates) can only be used in practice if users trust the authenticity of the CAs public keys
- For example, in the diagram below, how do A and B acquire the public key of the CA, and why / how can they trust this key?
- The CA is the **root of trust**, but how can this trust be justified?



# Direct Trust

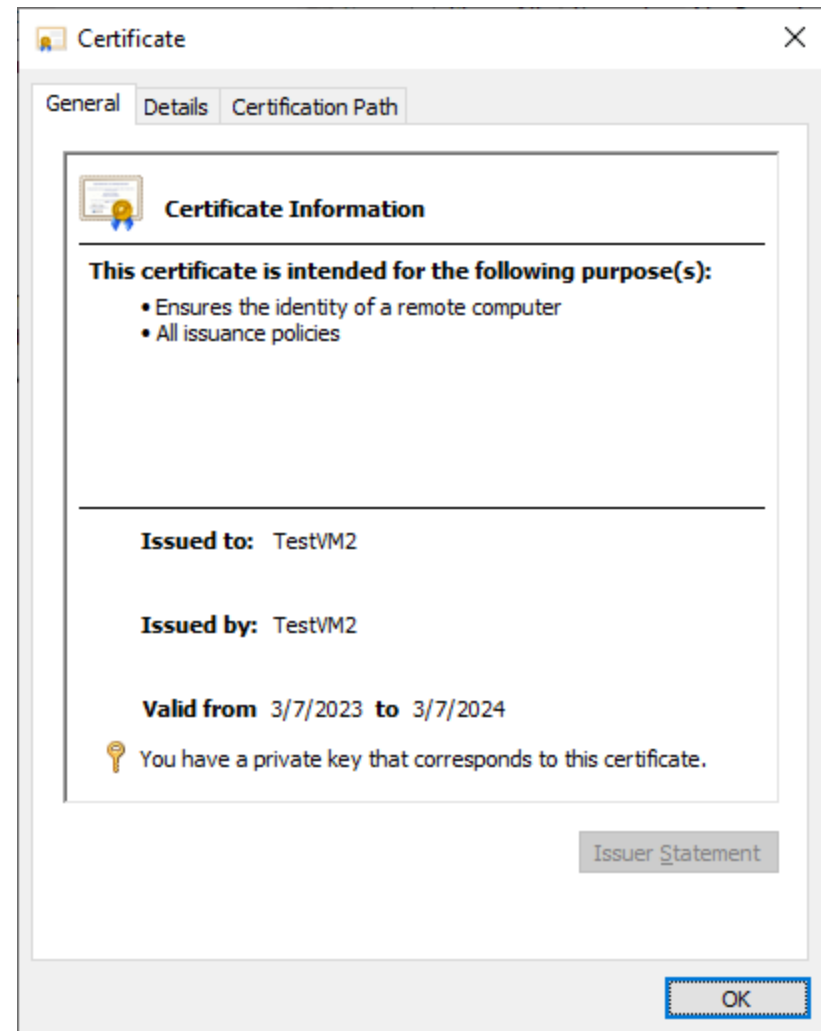
35

- Trust in the authenticity of a public key is direct if the public key is directly obtained from the key owner or its owner directly confirms the authenticity of the key in a way that is convincing for the user
- Example:
  - ▣ Most Linux systems allow the installation of additional software such as updates or services from trusted servers located on the Internet
  - ▣ The authenticity of those software packages is established by a digital signature
  - ▣ The verification of the signature requires a public key, which is embedded in the Linux distribution
  - ▣ The authenticity of this key is guaranteed by the authenticity of the Linux installation image
  - ▣ Such public keys are usually internally stored as **self-signed certificates**
  - ▣ Similarly, self-signed certificates can be found in web browsers

# Self-Signed Digital Certificates

36

- ❑ Self-signed digital certificates are issued by the public key owner themselves, as opposed to a certificate authority (CA) issuing them
- ❑ Subject and issuer fields point to the same identity and the cert is signed using the owner's private key
- ❑ Obviously, they do not provide any trust value per se
  - ❑ However, root CA have self-signed certificates (→ later)
- ❑ See also self-signed browser certificates using OpenSSL
  - ❑ [https://www.akadia.com/services/ssh\\_test\\_certificate.html](https://www.akadia.com/services/ssh_test_certificate.html)



# Commercial CAs

- Self-signed certificates have no value to 3<sup>rd</sup> parties, as different users that need to exchange their certs need a common root of trust
- This is achieved by hundreds of companies worldwide that provide digital certificates to clients
  - e.g. Verisign ([www.verisign.com](http://www.verisign.com)) and SSL ([www.ssl.com](http://www.ssl.com))
- These CAs form a CA hierarchy

Rank	Issuer	Usage	Market Share
1	IdenTrust	43.4%	48.9%
2	DigiCert	16.6%	18.7%
3	Sectigo (Comodo Cybersecurity)	13.8%	15.5%
4	Let's Encrypt	7.2%	8.2%
5	GoDaddy	5.4%	6.1%
6	GlobalSign	2.4%	2.7%

# Certificate Classes

38

- Certificate classes in digital certificates are typically encoded using specific OIDs within the certificate's extensions
- These classes can indicate different levels of validation and trust, such as
  - ▣ domain validation (DV)
  - ▣ organization validation (OV)
  - ▣ extended validation (EV)

# Certificate Classes

39

Certificate Type	Validation Level	Issuance Time	Use Case	Assurance Level
<b>Domain Validation (DV)</b>	Basic	Minutes	Personal websites, blogs, small businesses	Low, does not verify the identity of the subject
<b>Organization Validation (OV)</b>	Intermediate	Few days	Business websites, organizations	Medium, validates the subject's identity
<b>Extended Validation (EV)</b>	Highest	Several days to weeks	E-commerce sites, financial institutions, websites handling sensitive data	High, as the CA conducts a thorough vetting process, including verifying the legal, physical, and operational existence of the organization

# Domain-Validated Certificates

40

- Digital certificates are usually issued to websites
  - ▣ The public key in it is used to setup a secure connection between client browser and server (by negotiating a symmetric key -> later)
- Practically, many CAs often do not do a thorough check on a website (e.g. malware check) or their owners (id, credentials etc.)
- Instead, automatic checks are done, where it is validated that the applicant has control over the website and the DNS of the website domain, e.g.,
  - ▣ Place a specific file at the specific URL on the website
  - ▣ Add a specific DNS record to the website domain
  - ▣ Create an email address in the site domain and receive a password at that email
- As a result, such (HTTPS) certificates are called domain-validated certificates

# Certificate Signing Request (CSR)

41

- ❑ A CSR is a Base64-and BER-encoded message (formally described using ASN.1) sent from an applicant to a CA of the PKI in order to apply for a digital certificate
- ❑ The most common format for CSRs is the PKCS #10 specification
  - ▣ PKCS stands for "Public Key Cryptography Standards"
- ❑ Before creating a CSR, the applicant first generates a key pair, keeping the private key secret
- ❑ The CSR subsequently contains the public key, as well as the following fields (source: Wikipedia):

DN <sup>[2]</sup>	Information	Description	Sample
CN	Common Name	This is <a href="#">fully qualified domain name</a> that you wish to secure	*.wikipedia.org
O	Organization Name	Usually the legal name of a company or entity and should include any suffixes such as Ltd., Inc., or Corp.	Wikimedia Foundation, Inc.
OU	Organizational Unit	Internal organization department/division name	IT
L	Locality	Town, city, village, etc. name	San Francisco
ST	State	Province, region, county or state. This should not be abbreviated (e.g. West Sussex, Normandy, New Jersey).	California
C	Country	The <a href="#">two-letter ISO code</a> for the country where your organization is located	US
EMAIL	Email Address	The organization contact, usually of the certificate administrator or IT department	



# In-class Activity: Generating a Digital Certificate

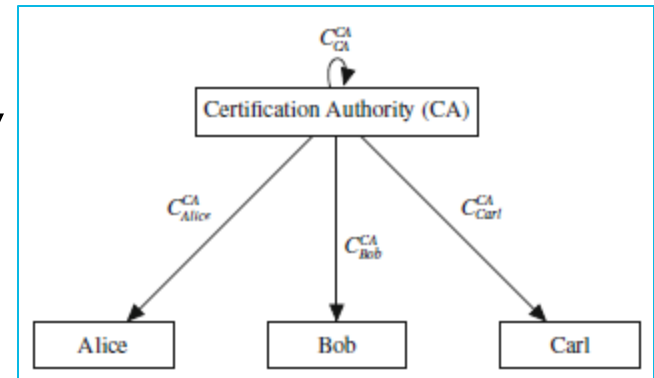
42

- Generate certificate signing request (CSR) via <https://csrgenerator.com/>
- View the CSR <https://lapo.it/asn1js/>
- Create a CSR and submit it to <https://getacert.com/>  
A certificate will be returned
- View the content of this certificate via
  - <https://lapo.it/asn1js/>
  - “Open in PEM format” in <https://getacert.com/>

# Hierarchical Trust

43

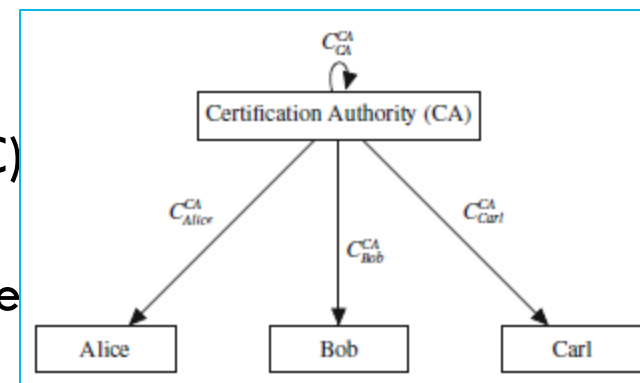
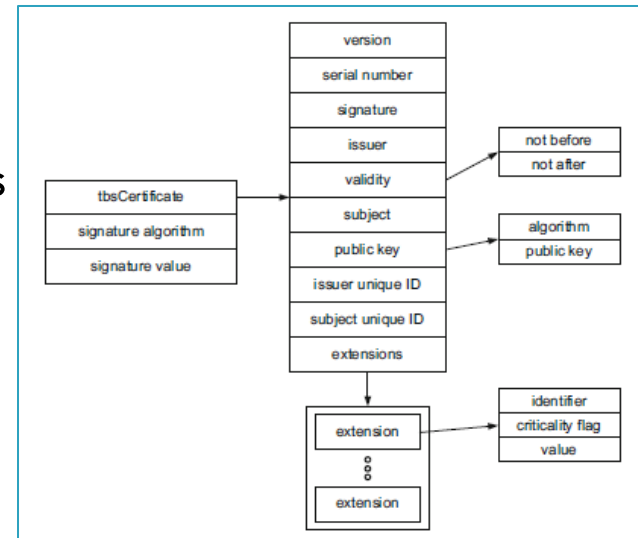
- In this simple hierarchical PKI, a single CA has issued certificates to the entities Alice, Bob, and Carl
- The CA is the trust anchor. It has generated a self-signed certificate, which is issued to Alice, Bob, and Carl too
  - ▣ The self-signing is depicted by a loop arrow from the CA to itself
- All entities in the PKI establish direct trust in the trust anchor
- Since the PKI users trust the trust anchor to sign certificates, the PKI users trust the authenticity of the public keys of Alice, Bob, and Carl, after validating their certificates
- Also, if entities outside the PKI trust the trust anchor and its public key, then they also accept the public keys of Alice, Bob, and Carl as authentic



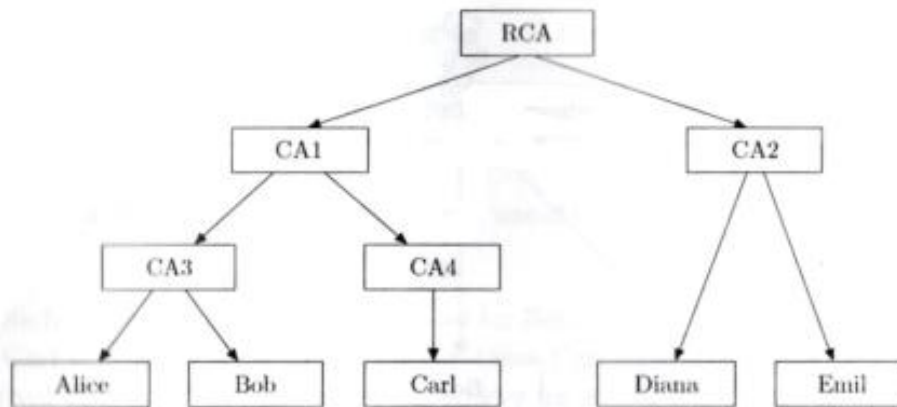
# Simple Hierarchical Trust Example

44

- Alice receives Bob's digital certificate (let's call it BDC) signed by the CA
- Alice checks the issuer section of BDC, which determines the CA being the issuer
- Alice has already a copy of the CAs self-signed certificate (let's call it CDC) and extracts the public key
  - Alice may even check the integrity of CDC in a similar way as she checks Bob's certificate below
- Alice validates that BDC has not expired
- She checks that the signature algorithm in BDC is compatible to CAs public key (e.g. RSA versus ECC)
- Alice decrypts BDC's signature value and compares it against the hash calculated over BDC excluding the signature value itself
- If both values match, the certificate and Bob's public key stored in it is valid
- Next, Alice validates Bob's authenticity via a challenge-response protocol

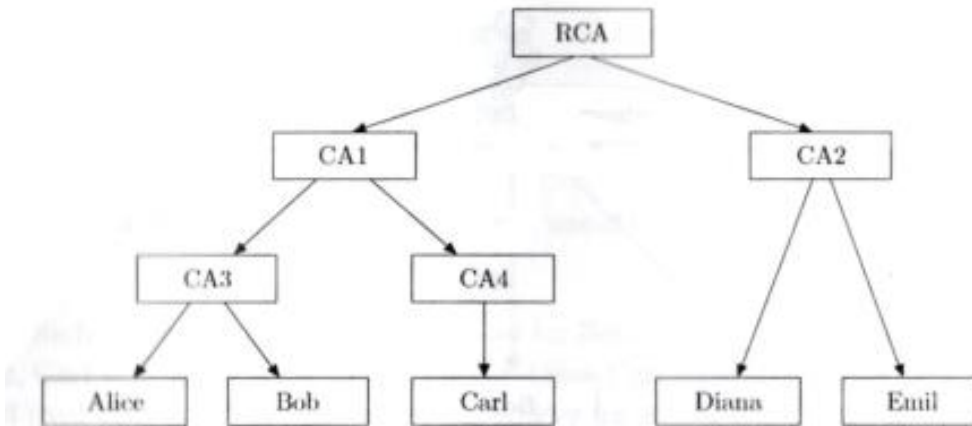


# CA Hierarchy I



- Assume a scenario, where multiple CAs provide certificates
- These CAs form a tree-like hierarchy with a “parent CA“ providing certificates for its “children”:
  - CA1 and CA2 are **intermediate** CAs whose certificates were signed by RCA
  - CA3 and CA4 are intermediate CAs whose certificates were signed by CA1
  - Alice and Bob have certificates signed by CA1
  - Carl’s certificate was signed by CA4
  - Diana’s and Emil’s certificate was signed by CA2
- Note that the leaves of this tree are **end-entities** (or end users)
- RCA could in principal sign end-entity certificates too
- End users and even CAs have no visibility of the entire CA hierarchy

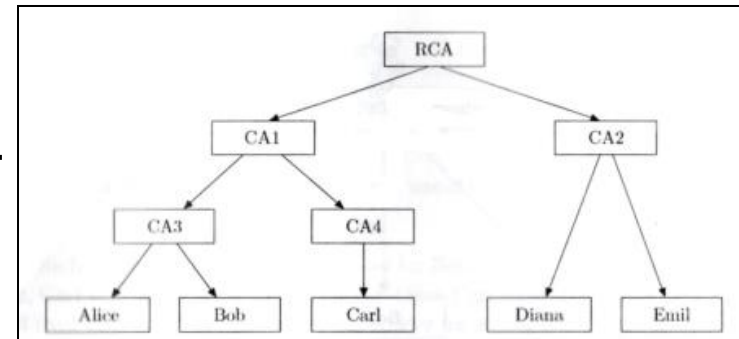
# CA Hierarchy II



- The RCA is the root of trust, and has a self-signed certificate
  - ▣ Remember that anybody could issue a self-signed cert to themselves!
- This RCA root certificate is distributed to all nodes in the hierarchy in a trustworthy fashion, for example via their
  - ▣ internet browser (a browser installation includes typically 200+ intermediate and root certificates) or
  - ▣ operating system installation

# CA Hierarchy III

- During operations, an endpoint may receive a certificate from another user that was signed by a CA unknown to them
  - E.g., Alice receives Emil's certificate that was signed by CA2
- Therefore, the user needs to get and validate the public key from an unknown CA (that is referenced in the received certificate), via a secure methodology, in order to validate the other user's certificate
  - E.g., Alice needs to acquire CA2's public key, and validate its authenticity, before validating Emil's certificate
- This process is called **Certification Path Construction**



# Certification Path Construction

48

- Consists of two phases:

- ▣ Path construction

- Involves building one or more *candidate* certification paths; "candidate" indicating that although the certificates may chain together properly, the path itself may not be valid for other reasons such as exceeding a maximum path length

- ▣ Path validation

- Involves making sure that each certificate in the path is within its established validity period, has not been revoked, and any constraints (e.g. maximum path length) are honoured

# Certification Path Construction via Name Chaining

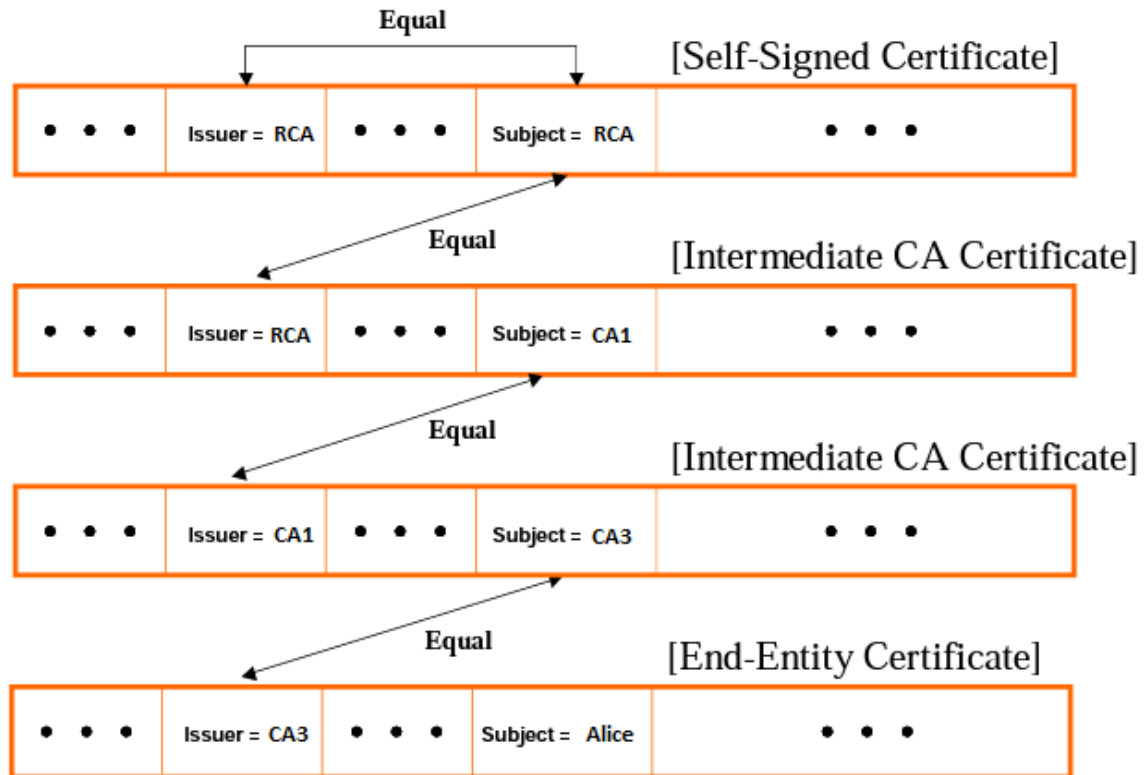
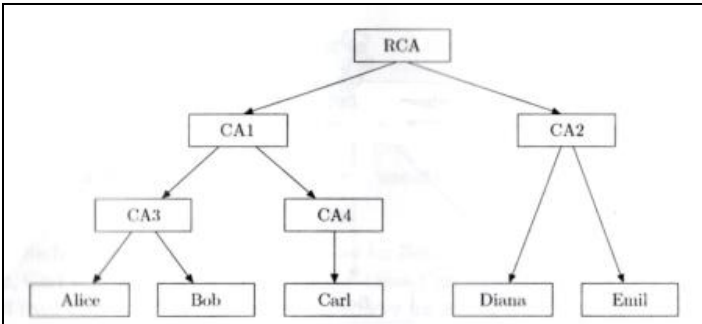
49

- A candidate certification path must "name chain" between the recognised trust anchor (example RCA) and the target (example Alice's) certificate
- Working from the trust anchor to the target certificate, this means that the Subject Name in one certificate must be the Issuer Name in the next certificate in the path, and so on



# Name Chaining Example

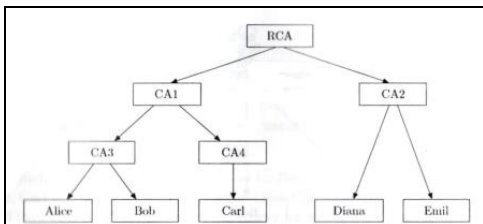
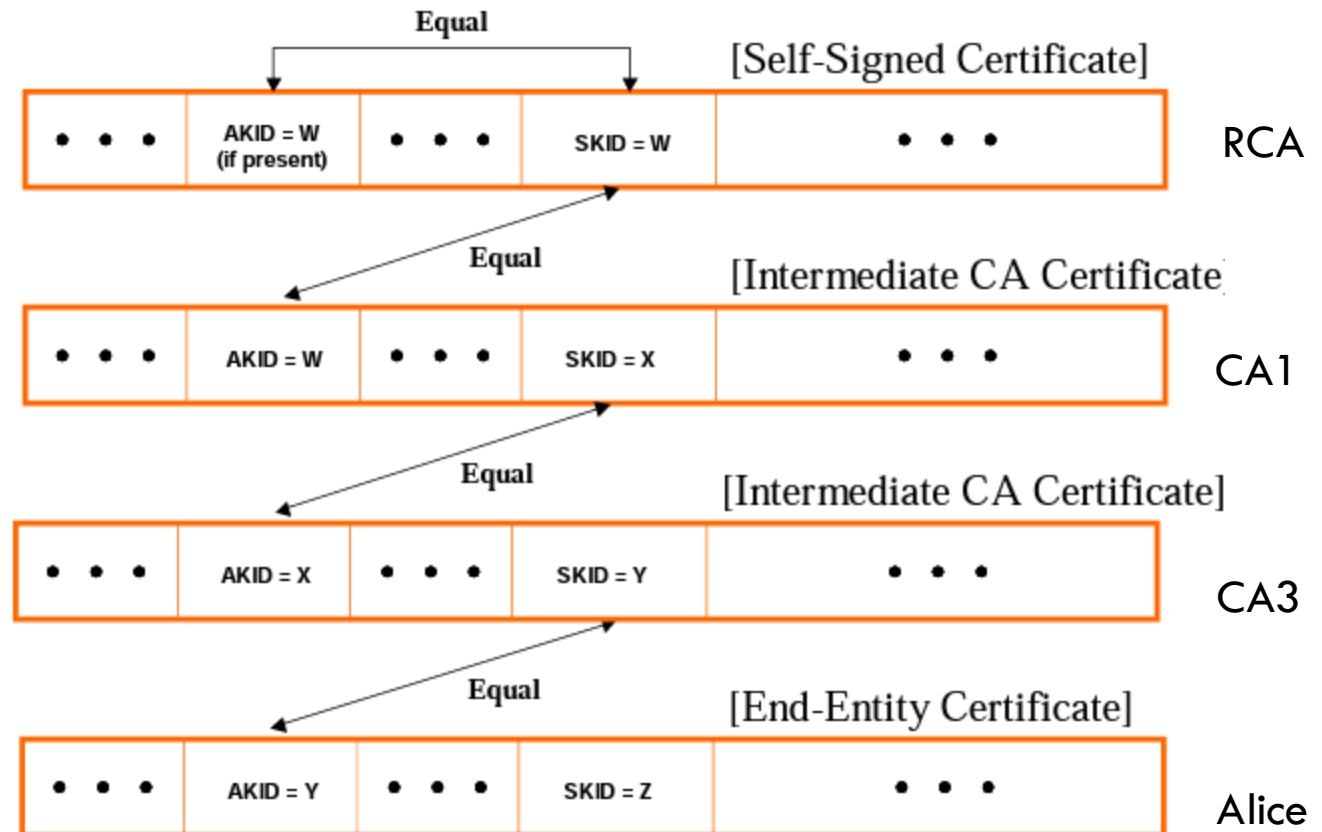
50



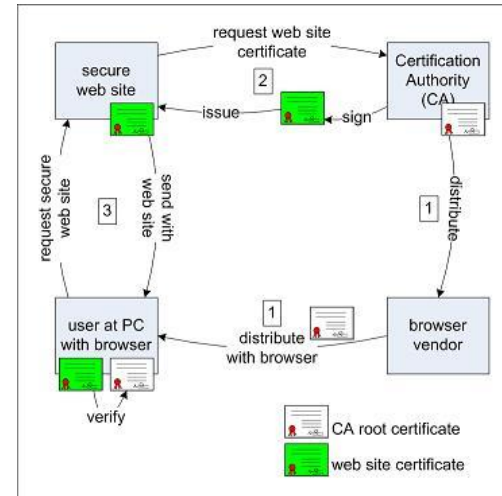
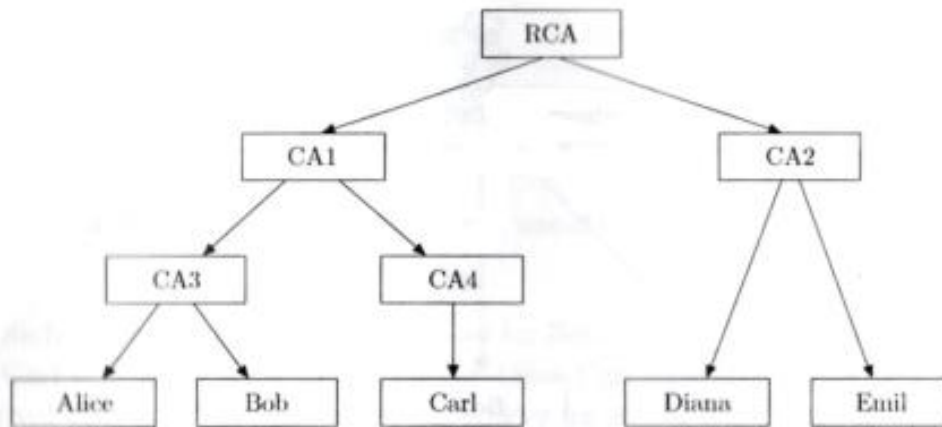
# Certification Path Construction via Key Identifier Chaining

51

- Recall certificate extensions *AuthorityKeyIdentifier* (AKID) and *SubjectKeyIdentifier* (SKID)



# Example Certificate Path Construction

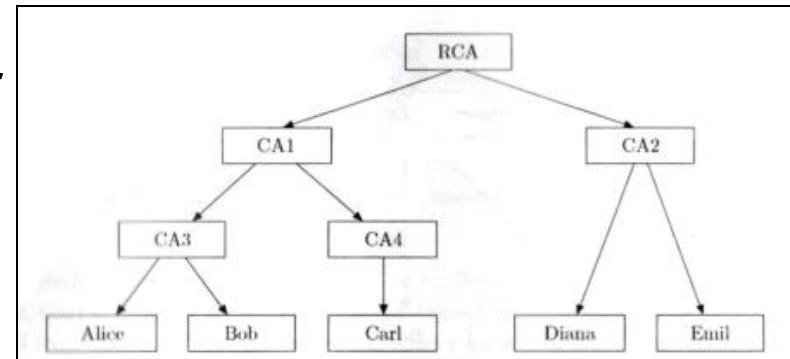


- Consider an example with
  - “Alice” (left) being “secure website” (right)
  - “Emil” (left) being “user at PC” (right)
  - “RCA” (left) being “Certificate Authority (CA)” (right)
- Emil sends a HTTPS connection request to Alice and receives a response containing her digital certificate
- Emil cannot validate Alice’s certificate directly, because it was signed by CA3 (and not RCA or CA2)
- However, if Emil can construct a Certification Path between Alice’s certificate and the RCA, he can validate Alice’s certificate (assuming he acknowledges the RCA as the root of trust)

# Certification Path Construction

53

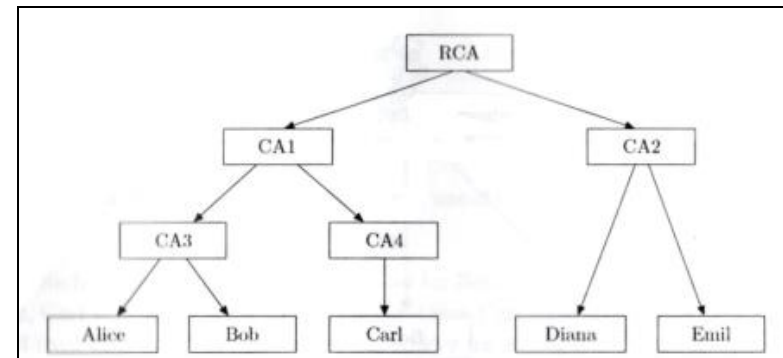
- In order for Emil to build the path, he must get copies of CA3's and CA1's certificates
  - ▣ RCA's self-signed cert is already in Emil's possession
- This can be done in 2 ways:
  1. Alice tags both certificates to hers and send all 3 of them to Emil
  2. Emil uses a directory service to retrieve both CA certificates, for example via LDAP (Lightweight Directory Access Protocol)



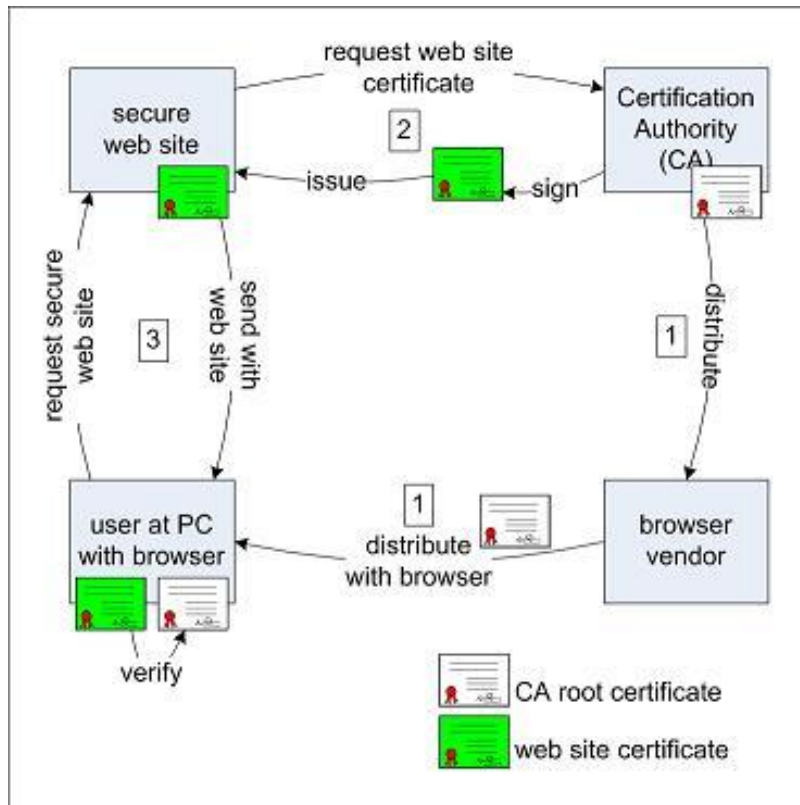
# Path Validation

54

- Now that Emil has a candidate path and all certificates, he must validate everything
  1. Firstly, Emil checks if all certificates have not expired yet (more later!)
  2. Then, using RCA's public key, he validates CA1's certificate as seen before
  3. If CA1's certificate is ok, Emil extracts its public key to validate CA3's certificate
  4. If CA3's certificate is ok, Emil extracts its public key to validate Alice's certificate
  5. If Alice's certificate is ok, and if her domain name (remember Alice is a secure website) matches the URL Emil entered, Emil goes ahead with the connection



# HTTPS Server Authentication Process (→ later)



- HTTPS is a secure version of HTTP
- In HTTPS, HTTP operates on top of TLS (Transport Layer Security), a secure transport layer protocol

# Basic Constraints

56

```
BasicConstraints ::= SEQUENCE {  
    cA                BOOLEAN DEFAULT FALSE,  
    pathLenConstraint INTEGER (0..MAX) OPTIONAL }
```

- ❑ Another X.509v3 extension...
- ❑ It is marked critical if the subject of the certificate is a CA
- ❑ cA is a Boolean value which is true if the certificate belongs to a CA and false otherwise
  - ▣ If this value is true, then the public key contained in the certificate can be used to verify signatures

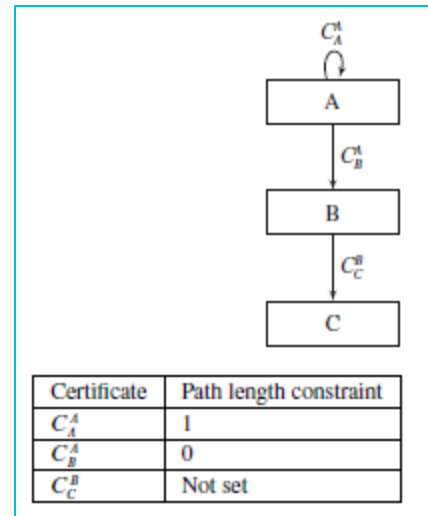
# Basic Constraints

57

□ It has two fields, the 2<sup>nd</sup> field:

```
BasicConstraints ::= SEQUENCE {  
  cA                BOOLEAN DEFAULT FALSE,  
  pathLenConstraint INTEGER (0..MAX) OPTIONAL }
```

- *pathLenConstraint* is used only for CA certificates in which the *cA* field is true and the *keyCertSign* bit is set in the key usage extension
- The value of this field is an integer; it sets a limit on the number of intermediate CA certificates that may be found after this certificate in the certification path before the path is invalid (i.e., when A generates B's certificate, it inserts its *pathLenConstraint* - 1
- Self-issued certificates do not count
- If such a limit is not desired, then this field is empty
- This parameter allows to limit the depth of a CA hierarchy



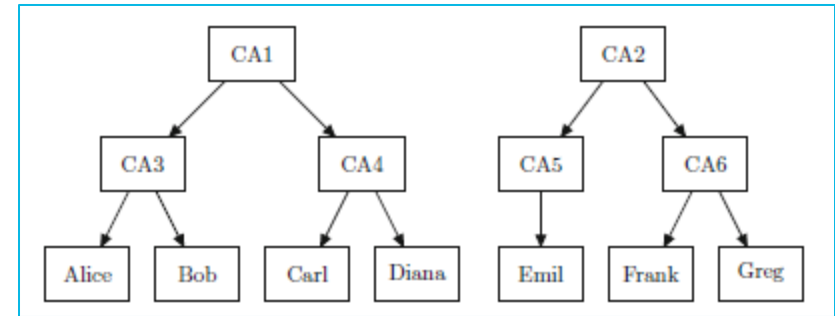


# Combining Trust Hierarchies: Trusted Lists

58

- Assume two independent PKIs with their own trust anchor
- How can Alice validate Greg's certificate?

- Solution 1: Trusted lists

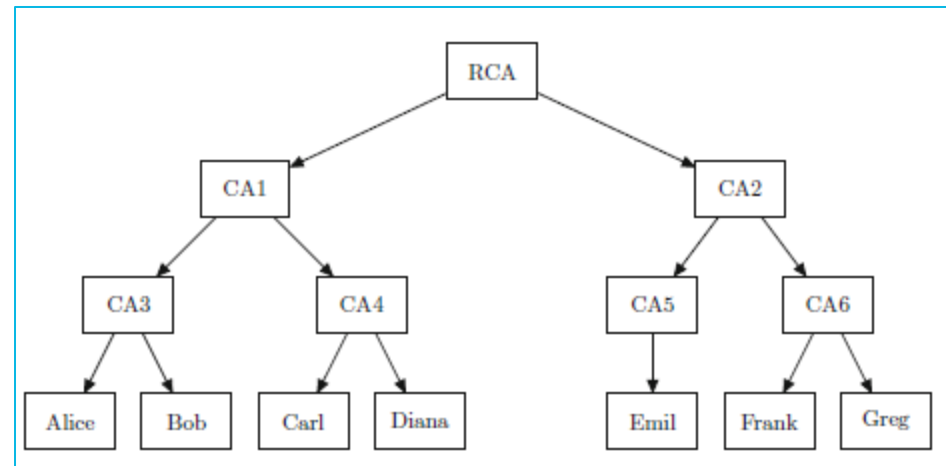


- Here Alice accepts CA2 as another trust anchor (note that her cert is signed by CA3 only)
  - CA2 cert is pre-installed on her browser / OS
- She is then able to construct a certification path (Greg – CA6 – CA2, potentially using a directory service), subsequently
  - validating CA6's cert using the public key in CA2's cert
  - validating Greg's cert using the public key in CA6's cert

# Combining Trust Hierarchies: Provide a common Root

59

- Here each end entity of the combined PKIs replaces its original trust anchor by the new common root

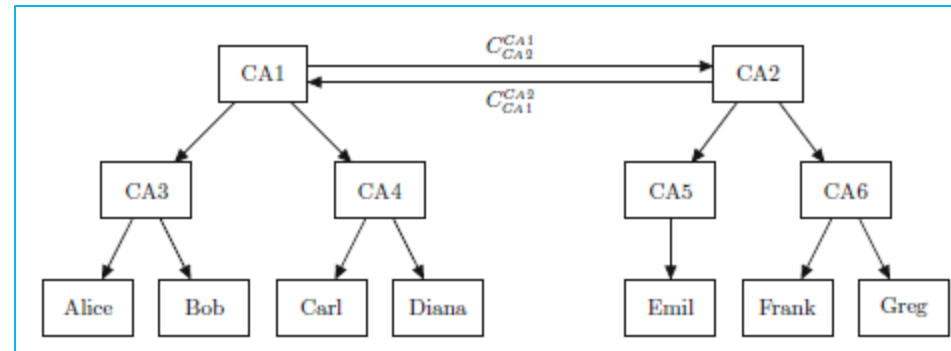


- As a consequence, certification paths that establish the authenticity of a public key have to be changed by prepending the common root

# Combining Trust Hierarchies: Cross Certification

60

- Cross-certification allows users of two PKIs to authenticate each other's public keys without replacing their trust anchors
- The idea is that the two root CAs certify each other's public keys using so-called **cross-certificates**
- In fact, the two CAs that cross-certify each other may also be only intermediate CAs
  - ▣ However, this implies that only the users covered by these CAs can validate each other's public keys
  - ▣ E.g. a single cross-certificate between CA4 and CA5 provides only interoperability between Carl, Diana and Emil



# Certificate Revocation

62

- The validity period of certificates may be quite long
  - ▣ For example, X.509 server certificates issued by SSL are typically valid for at least 2 years
- However, it may happen that during the validity period a certificate has to be invalidated
  - ▣ Example: the private key that corresponds to the public key in the certificate has been compromised
- The process of invalidating the certificate before its expiration time is called revocation

# Certificate Revocation Lists (CRL)

63

- A CRL is a list of revoked certificates which is digitally signed to prove its authenticity
- CRLs are regularly updated and made available at predictable points in time
  - ▣ When a CRL is updated, newly revoked certificates are inserted into the CRL
- There are direct CRLs and indirect CRLs:
  - ▣ Direct CRLs only contain certificates of one issuer and are issued and signed by that issuer
  - ▣ Indirect CRL may contain certificates of several issuers and is signed by the so-called CRL issuer
- Users who wish to obtain revocation information
  - ▣ download the CRL and verify its digital signature
  - ▣ check whether the certificate that they are interested in is contained in the CRL
- CRLs may become quite large since expired certificates are not always removed
- Therefore, delta CRLs have been introduced which only contain the certificates that have been revoked after the publication of the last full CRL
- The full CRL (i.e. complete CRL) contains all revoked certificates

# Online Certificate Status Protocol (OCSP)

64

- ❑ CRLs may become very large, downloading them becomes time consuming, and storing may need a lot of (unavailable) space
- ❑ Also, due to the potentially long time intervals between the publication of two subsequent lists, revocation information may not be up to date when it is used, in particular, shortly before the next update
- ❑ OCSP allows clients to query an OCSP server about the revocation status of individual certificates
- ❑ Here users may obtain revocation information immediately after the certificate is revoked
  - ▣ Unless of course the server just queries a CRL
- ❑ OCSP responses are digitally signed by the OCSP server, so they can be validated for their authenticity
- ❑ On the other hand, in contrast to the CRL method, OCSP requires the applications that need revocation information to be online

# Validity Models for Digital Signatures

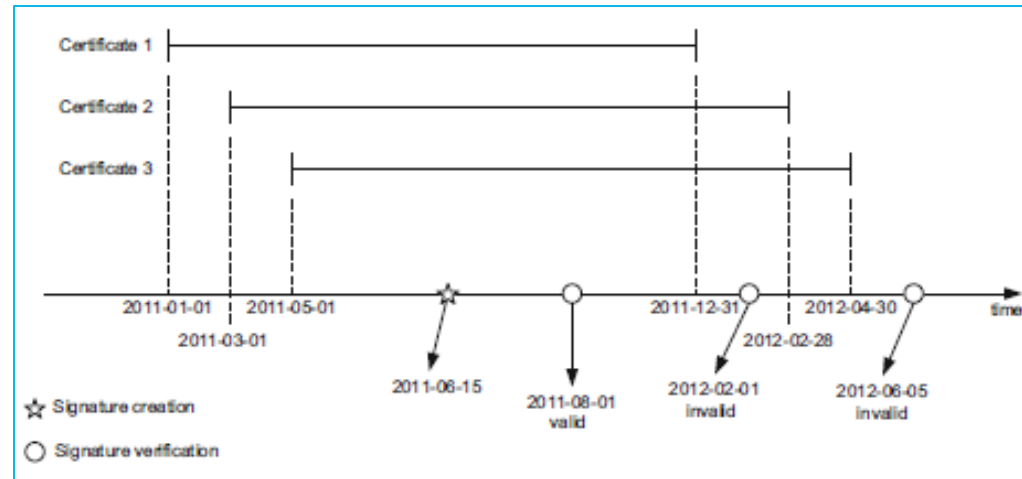
65

- Certificates in a validation path may have different expiry dates (because they were generated by different entities with different policies at different times), which poses the question, for how long an end-user certificate may be deemed valid, i.e. when does its path validation invalidate
- Simple example:
  - ▣ Assume Paul sells his house to Anna on 1 October 2023
  - ▣ Paul signs the sales contract digitally
  - ▣ The certificate that authenticates Paul's signature verification key expires on 31 July 2024
  - ▣ Should Paul's signature still be considered valid after his certificate has expired?

# The Shell Model

66

- In this model all certificates along the certification path must be valid when the signature is checked
- This model is appropriate in all applications, where signing and verification times are very close to each other
- Examples of such applications are
  - ▣ challenge-response authentication
  - ▣ mechanisms or email authentication
- However, for contract signing (with a legal binding long into the future) this model is inappropriate

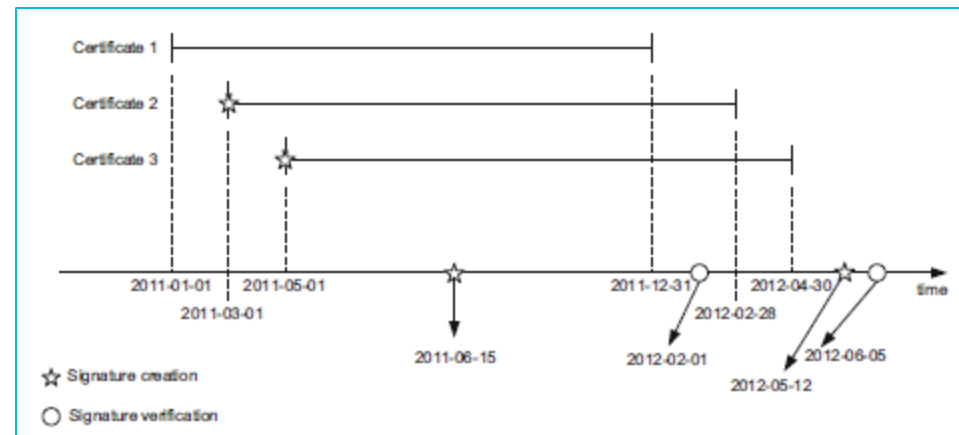




# The Chain Model

67

- In the chain model the validity of a signature is independent of the verification time for this signature
- The chain model is often used for verifying legally binding electronic signatures because such signatures may be used for contract signing
- The chain model supports long validity periods for digital signatures
- However, it has certain drawbacks:
  - ▣ If Alice issues a signature and later a certificate in the chain that certifies Alice's verification key is revoked, the signature remains valid
  - ▣ This may have serious effects if the revocation reason is key compromise
- In the above example, the "2011-06-01" signature is valid at the point "2012-06-06", the signature "2012-05-12" is not



# PKI Architecture Components

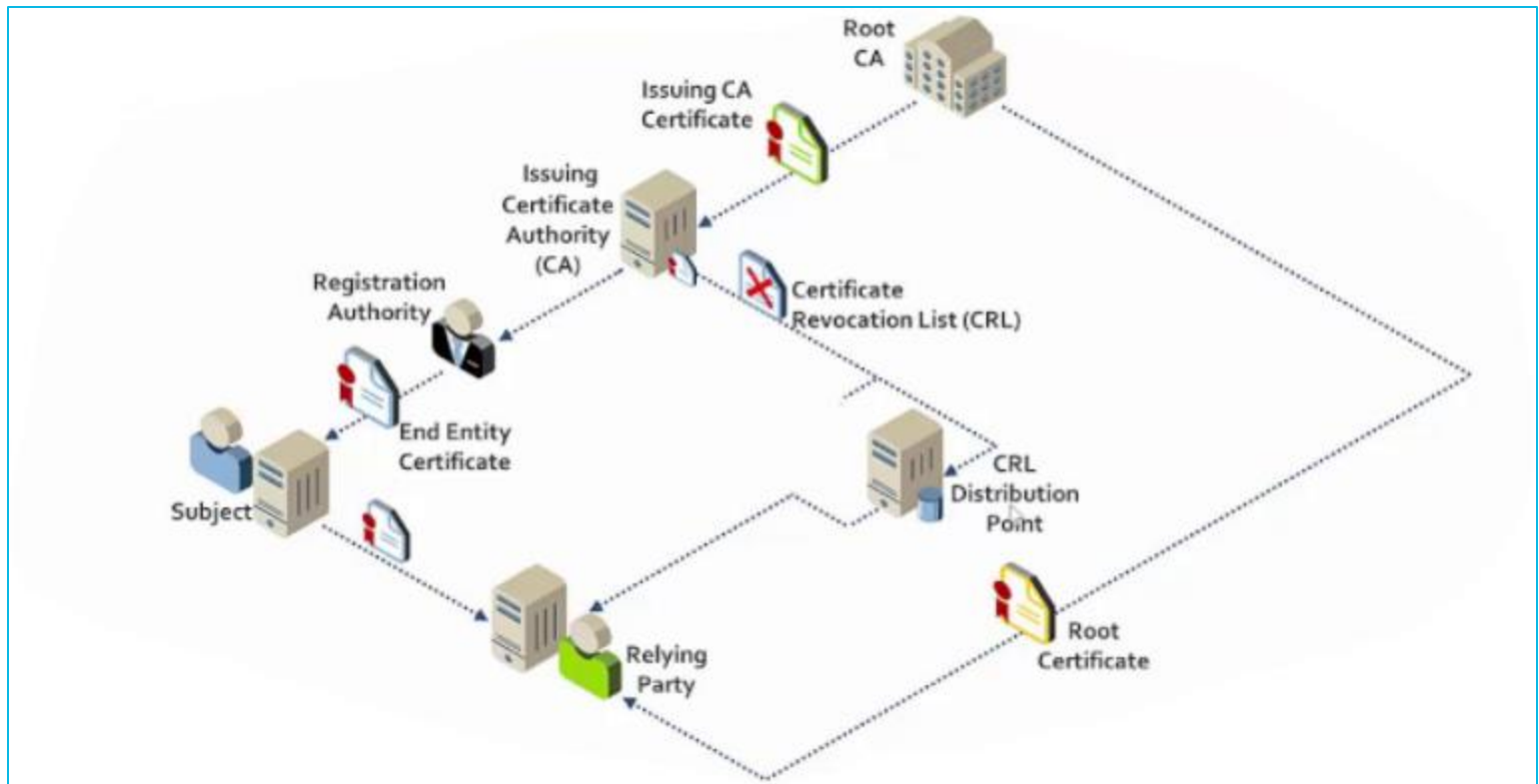
68

- ❑ A CA is a very well protected infrastructure that should only generate / sign certificates and CRLs
- ❑ Often, a RCA is only turned on on-demand (as a means of protecting it against attacks) to generate certificates for intermediate CA
- ❑ Such intermediate CA do all the signing work
  - ▣ It accepts CSRs (as seen before) from clients
- ❑ However, in order to reduce the attack surface of such a CA, client / end user communication including the processing of CSR, is done by a registration authority (RA)
- ❑ Similarly, CRL are distributed via dedicated CRL distribution points

# Example for a PKI Architecture

69

- Putting all components together, results in an architecture as shown below
- The Relying Party may be a web browser
- The Subject may be a web server



# FYI: ASN.1

70

- Abstract Syntax Notation One (ASN.1) is a standard interface description language for defining data structures that can be serialised and de-serialised in a cross-platform way
- Originally introduced to describe network data packets exchanged between endpoints, it is also widely used in cryptography and biometrics
- It is closely associated with a set of encoding rules that specify how to represent a data structure as a series of bytes, i.e.,
  - ▣ Basic Encoding Rule (BER)
  - ▣ Distinguished Encoding Rules (DER)
- Here encoded elements are typically type-length-value (TLV) sequences

# FYI: ASN.1 Basic Syntax

71

- ASN.1 is case sensitive
- Keywords start with capital letter
- Comments start with "--"
- The underscore is forbidden in identifiers and keywords
- Assignments use symbol "::="
- The top-level container of a type declaration is a module, e.g.

```
myModule DEFINITIONS ::= BEGIN
```

```
...
```

```
END
```

# FYI: ASN.1 Basic Syntax

72

- The available basic types are:
  - BOOLEAN
  - INTEGER
  - ENUMERATED
  - REAL
  - NULL
- Examples:
  - Automatic ::= BOOLEAN
  - Color ::= ENUMERATED {red, blue, green}
  - Pi REAL ::= 3.141
- Important: All types are abstract, e.g. there is no length or size associated with an INTEGER
- There are 3 types of strings (character, binary and hexadecimal), e.g.
  - IA5STRING ::= "Hello World" – International alphabet 5 with 7-bit characters
  - encryptionKey BIT STRING ::= '00100'B
  - encryptionKey OCTET STRING ::= 'ABC01'H

# FYI: ASN.1 Restricted Types

73

- Range:
  - ▣ Example: Age ::= INTEGER (0..50)
- Value set:
  - ▣ Example: Age ::= INTEGER {5, 10, 15, 20}
- Enumerated values
  - ▣ Example: Color ::= ENUMERATED {red(1), blue(2)}
- Default type
  - ▣ Example: Age ::= INTEGER DEFAULT 42

# FYI: ASN.1. Structured Types

74

- SEQUENCE
  - ▣ Like a struct in C
  - ▣ Example: See next slide
- SEQUENCE OF
  - ▣ Sequence of the same type
  - ▣ Example: `myCars ::= SEQUENCE OF Car`
- SET
  - ▣ Like a set
- SET OF
  - ▣ Set of the same type
- CHOICE
  - ▣ Similar to a union in C



# Example ASN.1 (Wikipedia)

75

Consider the following ASN.1 definition:

```
FooProtocol DEFINITIONS ::= BEGIN
  FooQuestion ::= SEQUENCE {
    trackingNumber INTEGER(0..199),
    question      IA5String
  }
  FooAnswer ::= SEQUENCE {
    questionNumber INTEGER(0..199),
    answer          BOOLEAN
  }
  FooHistory ::= SEQUENCE {
    questions SEQUENCE(SIZE(0..10)) OF FooQuestion,
    answers   SEQUENCE(SIZE(1..10)) OF FooAnswer,
    anArray   SEQUENCE(SIZE(100)) OF INTEGER(0..1000),
    ...
  }
END
```

Example for FooQuestion:

```
FooQuestion ::= SEQUENCE {
  trackingNumber INTEGER(5),
  question      "Anybody there?"
}
```

ASN.1 description of a simple application layer question / response protocol between a client and a server

# ASN.1 Encoding Formats

76

- There are three ASN.1 encoding formats:
  - ▣ Basic Encoding Rules (BER)

The original rules laid out by the ASN.1 standard for encoding data into a binary format
  - ▣ Canonical Encoding Rules (CER)
  - ▣ Distinguished Encoding Rules (DER)
- Both CER and DER are subsets of BER
  - ▣ Whereas BER gives choices as to how data values may be encoded, CER (together with DER) selects just one encoding from those allowed by the basic encoding rules
    - For example: In BER a Boolean value of true can be encoded as any positive integer up to 255, while in DER it has to be a 1

# BER Overview

- ❑ BER specifies a self-describing and self-delimiting format for encoding ASN.1 data structures
- ❑ Each data element is encoded as a type identifier, a length description, the actual data elements (TLV format), and, where necessary, an end-of-content marker
  - ❑ These types of encodings are commonly called type-length-value (TLV) encodings



First length octet



Form	Bits							
	8	7	6	5	4	3	2	1
Definite, short	0	Length (0–127)						
Indefinite	1	0						
Definite, long	1	Number of following octets (1–126)						
Reserved	1	127						



Long form example, length 435

	Octet 1								Octet 2								Octet 3							
	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	1	1
Long form	2 length octets								110110011 <sub>2</sub> = 435 <sub>10</sub> content octets															

# Some BER Identifier Octets and their Encodings (Wikipedia)

Name	Permitted construction	Tag number	
		Decimal	Hexadecimal
End-of-Content (EOC)	Primitive	0	0
BOOLEAN	Primitive	1	1
INTEGER	Primitive	2	2
BIT STRING	Both	3	3
OCTET STRING	Both	4	4
NULL	Primitive	5	5
OBJECT IDENTIFIER	Primitive	6	6
Object Descriptor	Both	7	7
EXTERNAL	Constructed	8	8
REAL (float)	Primitive	9	9
ENUMERATED	Primitive	10	A
EMBEDDED PDV	Constructed	11	B
UTF8String	Both	12	C
RELATIVE-OID	Primitive	13	D
TIME	Primitive	14	E

□ The identifier octets encode the ASN.1 tag's class number and type number

Octet 1							Octet 2 ... n Only if tag type > 30 <sub>10</sub>								
8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
Tag class			P/C		Tag type (if 0-30 <sub>10</sub> )			Long Form							
					31 <sub>10</sub> = Long Form			1=More		7 bits of Tag type					

Normally all 0

Identifier octets <i>Type</i>	Length octets <i>Length</i>	Contents octets <i>Value</i>	End-of-Contents octets <i>(only if indefinite form)</i>
----------------------------------	--------------------------------	---------------------------------	--

# Example BER Encoding (Wikipedia)

79

Consider the following ASN.1 definition:

```
FooProtocol DEFINITIONS ::= BEGIN
  FooQuestion ::= SEQUENCE {
    trackingNumber INTEGER(0..199),
    question      IA5String
  }
  FooAnswer ::= SEQUENCE {
    questionNumber INTEGER(10..20),
    answer          BOOLEAN
  }
  FooHistory ::= SEQUENCE {
    questions SEQUENCE(SIZE(0..10)) OF FooQuestion,
    answers   SEQUENCE(SIZE(1..10)) OF FooAnswer,
    anArray  SEQUENCE(SIZE(100)) OF INTEGER(0..1000),
    ...
  }
END
```

The FooQuestion structure “5Anybody there?” encoded in DER format:

```
30 13 02 01 05 16 0e 41 6e 79 62 6f
64 79 20 74 68 65 72 65 3f
```

with

- 30 — type tag indicating SEQUENCE
- 13 — length in octets of value that follows
- 02 — type tag indicating INTEGER (see previous slide)
- 01 — length in octets of value that follows
- 05 — value (5)
- 16 — type tag indicating IA5String (i.e. ASCII)
- 0e — length in octets of value that follows
- 41 6e 79 62 6f 64 79 20 74 68 65 72 65 3f (“Anybody there?” in plain ASCII format)

# ASN.1 Encoding of OIDs

80

- Practically, OIDs need to be encoded as TLV triplets
- The TLV triplet begins with a Tag value of 0x06 (see table on the right)
- Each OID integer (i.e., node) is encoded as follows:
  - ▣ The first two nodes of the OID are encoded onto a single byte, by multiplying the first node with 40 and adding the result to the value of the second node
  - ▣ Subsequent bytes are represented using **Variable Length Quantity**, also called base 128

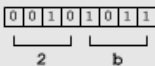
Name	Permitted construction	Tag number	
		Decimal	Hexadecimal
End-of-Content (EOC)	Primitive	0	0
BOOLEAN	Primitive	1	1
INTEGER	Primitive	2	2
BIT STRING	Both	3	3
OCTET STRING	Both	4	4
NULL	Primitive	5	5
OBJECT IDENTIFIER	Primitive	6	6
Object Descriptor	Both	7	7
EXTERNAL	Constructed	8	8
REAL (float)	Primitive	9	9
ENUMERATED	Primitive	10	A
EMBEDDED PDV	Constructed	11	B
UTF8String	Both	12	C
RELATIVE-OID	Primitive	13	D
TIME	Primitive	14	E

# Example: BER Encoding of an OID




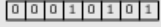
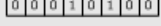
81

**OID:**  
1.3.6.1.4.1.311.21.20 (ClientId Attribute)

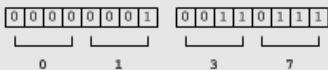
**1) Encoding the First Two Nodes:**

$1 \times 40 + 3 = 43d = 0x2B =$  

**2) Single byte encoding of all remaining nodes other than 311:**

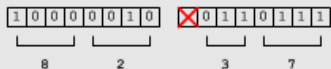
3 = 0x03 =   
 6 = 0x06 =   
 4 = 0x04 =   
 21 = 0x15 =   
 20 = 0x14 = 

**3) Multiple byte encoding of 311:**

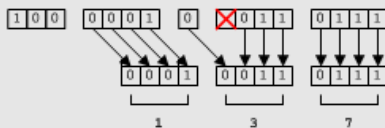
311d = 0x0137 = 

This is encoded to 0x82 0x37 by:

- Setting bit 7 of the leftmost byte to 1.
- Ignoring bit 7 of the rightmost byte.
- Shifting the right nibble of the leftmost byte to the left by 1 bit.



When decoded, the bits are assembled in the following manner:



**3) Summary encoding of OID : 1.3.6.1.4.1.311.21.20**

0x2B 0x06 0x01 0x04 0x01 0x82 0x37 0x15 0x14

- This example shows how the *ClientId* attribute (OID: 1.3.6.1.4.1.311.21.20) of a certificate request is encoded:  
1.3.6.1.4.1.311.21.20vich3d.jdomcsc.nettest.microsoft.comJDOMCSCadministratorcertreq”

```

06 09 ; OBJECT_ID (9 Bytes)
| 2b 06 01 04 01 82 37 15 14 ; 1.3.6.1.4.1.311.21.20
31 4a ; SET (4a Bytes)
 30 48 ; SEQUENCE (48 Bytes)
  02 01 ; INTEGER (1 Bytes)
  | 09
 0c 23 ; UTF8_STRING (23 Bytes)
  | 76 69 63 68 33 64 2e 6a ; vich3d.j
  | 64 6f 6d 63 73 63 2e 6e ; domcsc.n
  | 74 74 65 73 74 2e 6d 69 ; ttest.mi
  | 63 72 6f 73 6f 66 74 2e ; crossoft.
  | 63 6f 6d ; com
 0c 15 ; UTF8_STRING (15 Bytes)
  | 4a 44 4f 4d 43 53 43 5c ; JDOMCSC\
  | 61 64 6d 69 6e 69 73 74 ; administ
  | 72 61 74 6f 72 ; rator
 0c 07 ; UTF8_STRING (7 Bytes)
  63 65 72 74 72 65 71 ; certreq
    
```

# Base64 Encoding

82

- Problem: How can BER encoded binary data (including certificates) be stored or transported in channels that only reliably support (readable) text content?
- Examples:
  - ▣ Embedding (binary) images inside textual assets such as HTML and CSS files
  - ▣ Embedding attachments (e.g. images) in emails
- Solution: Apply a binary-to-text encoding scheme, e.g. Base64



# Base64 Encoding

- Base64 divides a binary input into 6-bit snippets, with each snippet represented by a printable character
- Example Base64 table from RFC 4648:

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

# Base64 Encoding Examples (Wikipedia)

84

- “Many hands make light work” is converted into  
TWFueSBoYW5kcyBtYWtlIGxpZ2h0IHdvcmsu
- Generally, 3 bytes are converted into 4 printable  
Base64 characters (with padding character “=” added  
if input length is not multiple of 3), as follows:

Source	Text (ASCII)	M			a			n																	
	Octets	77 (0x4d)			97 (0x61)			110 (0x6e)																	
Bits		0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Base64 encoded	Sextets	19			22			5			46														
	Character	T			W			F			u														
	Octets	84 (0x54)			87 (0x57)			70 (0x46)			117 (0x75)														

# Example: Base64 Encoded Certificate Signing Request (more later)

85

```
-----BEGIN NEW CERTIFICATE REQUEST-----  
MIICkzCCAXsCAQAwTjELMAkGA1UEBhMCQ0ExCzAJBgNVBAGTAmdmMQswCQYDVQQH  
EwJnZjELMAkGA1UECxmCZ2YxCzAJBgNVBAoTAmdmMQswCQYDVQQDEwJnZjCCASlw  
DQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMlwsZXhim1CYsCcz5MOwHlLhkxU  
3KAEhr1pg3tOPmzImuXTnWWt4sDb//fsadcZ9EBInUMoRurTLLo8TuNnNhAlkGD0  
9PPSEZPb+loYLASA8DG4SkRyrl2sVhIVmzq8w7/zp561ur5m3wV+c5ru3W/CvjdT  
Z78ReIUTlul2nCJ46PQIYky+2IPGtj+VY/9IDe+iXLs9i/u7k2oppBo70qdzR3vR  
hml55noblmeUcVL21w2jMTz6nZAnsatt4fnrAgM6ZmNzXyaoj3PNWoBYtSBuiYe  
QArBhiOpR1Og9E2XGOvbsyc4+ORNWPSfX0H4uFYZNAS5n4fBrFTskJ9MKEUCAwEA  
AaAAMA0GCSqGSIb3DQEBAQUAA4IBAQCTLS7EWjqVewqrotQ5NZa8IXIFSoGaNOeU  
WVJoXWUIkhd6CSOgxXiDdYIDIVe1EUGUQ5Lx9bVnniBy0F7ssUFBgehG6maxWrq7  
AEPFQESgfsEYH6JGvhZM1Fa9WjxaCiOXpozP1SIF9j6RzNvJudxpDOd80RSjoifg  
f4QXNfdW1fpXa56ED2NBgozXb1IWeu/Kb2JU7AlUmY6Xde1tAyW5I7gIbFapAacv  
//edvQZm1Zfq0/CVSKhxwcg8K8gf1rLfgTNPz7FbvGhDO9YFir7qVK1xx7HEaBe9  
BkQqxArSzTCtKpFbNPQ+A6mxBnVXXFhEOtNeaU/foq0k7l+3k9LD  
-----END NEW CERTIFICATE REQUEST-----
```

□ See <http://lapo.it/asn1js/>