Introduction to Cryptography
Lecture 10

Digital signatures,
Public Key Infrastructure (PKI)

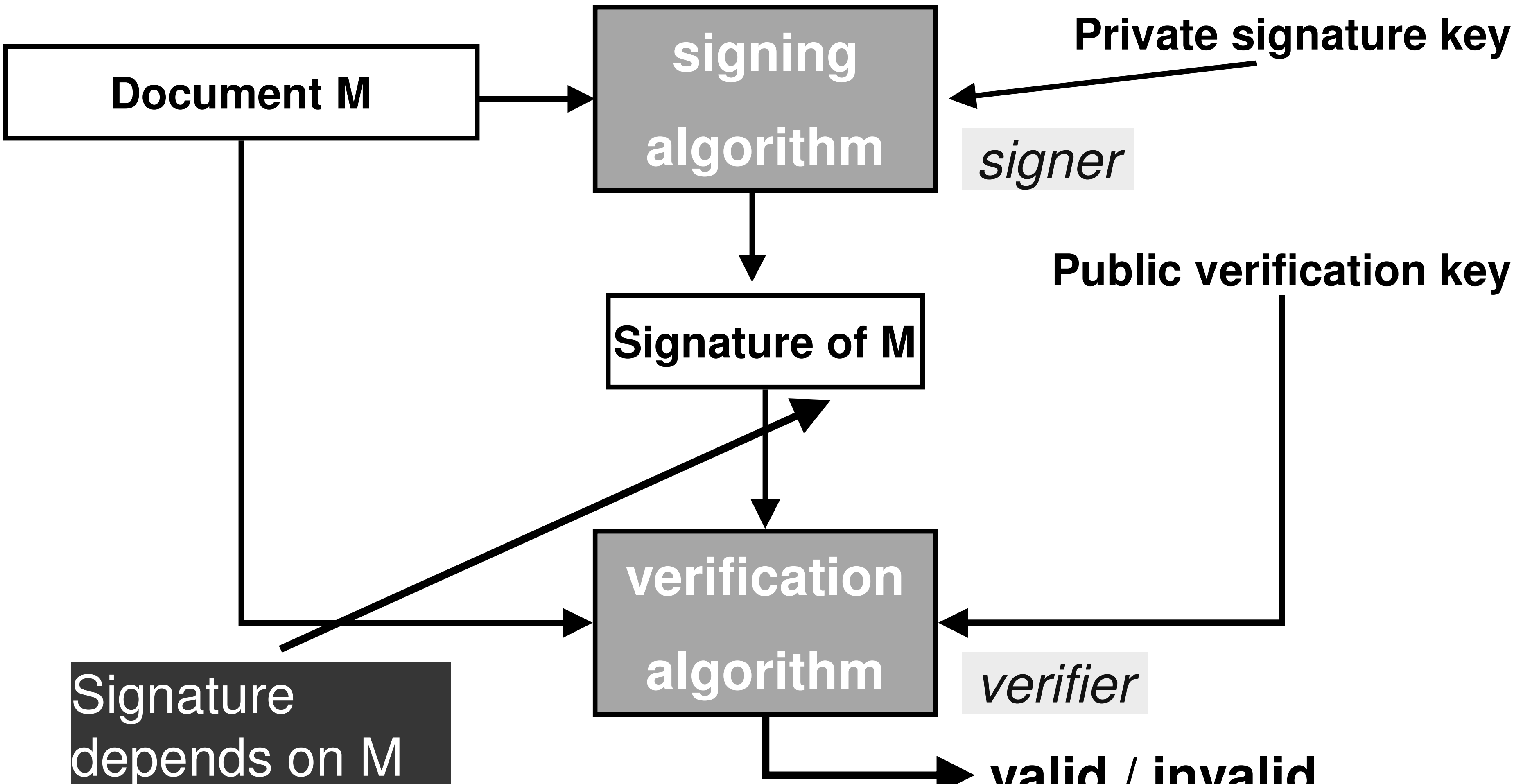
Benny Pinkas

Non Repudiation

- Prevent signer from denying that it signed the message
- I.e., the receiver can prove to third parties that the message was signed by the signer

- This is different than message authentication (MACs)
  - There the receiver is assured that the message was sent by the receiver and was not changed in transit
  - But the receiver cannot prove this to other parties
    - MACs: sender and receiver share a secret key *K*
    - If R sees a message MACed with *K*, it knows that it could have only been generated by S
    - But if R shows the MAC to a third party, it cannot prove that the MAC was generated by S and not by R

Signing/verification process



Message lengths

- A technical problem:
  - |m| might be longer than |N|
  - *m* might not be in the domain of $f^{-1}()$

Solution "hash-and-sign" paradigm:

- Signing: First compute *H(m)*, then compute the signature $f^{-1}(H(M))$. Where,
  - The range of *H()* must be contained in the domain of $f^{-1}()$.
  - *H()* must be collision intractable. I.e. it is hard to find (in polynomial time) messages *m, m'* s.t. *H(m)=H(m').*
- Verification:
  - Compute *f(s).* Compare to *H(m).*

- Using *H()* is also good for security reasons. See below.

Security definitions for digital signatures

- Attacks against digital signatures
  - *Key only attack:* the adversary knows only the verification key
  - *Known signature attack:* in addition, the adversary has some message/signature pairs.
  - *Chosen message attack:* the adversary can ask for signatures of messages of its choice (e.g. attacking a notary system).

    (Seems even more reasonable than chosen message attacks against encryption.)

Security definitions for digital signatures

- Several levels of success for the adversary
  - *Existential forgery:* the adversary succeeds in forging the signature of one message.
  - *Selective forgery:* the adversary succeeds in forging the signature of one message of its choice.
  - *Universal forgery:* the adversary can forge the signature of any message.
  - *Total break:* the adversary finds the private signature key.

- Different levels of security, against different attacks, are required for different scenarios.

Example: simple RSA based signatures

- Key generation: (as in RSA)
  – Alice picks random *p,q*. Defines $N=pq$ and finds $e \cdot d=1$ mod $(p-1)(q-1)$.
  – Public verification key: *(N,e)*
  – Private signature key: *d*

- Signing: Given *m*, Alice computes $s=m^d$ mod *N.*
- *(suppose that there is no hash function H())*

- Verification: given *m,s* and public key *(N,e).*
  – Compute $m' = s^e$ mod *N*.
  – Output "valid" iff m'=m.

Attacks against plain RSA signatures

- Signature of *m* is $s=m^d \bmod N$.
- Universally forgeable under a chosen message attack:
  - *Universal forgery:* the adversary can forge the signature of any message of its choice.
  - *Chosen message attack:* the adversary can ask for signatures of messages of its choice.
- Existentially forgeable under key only attack.
  - *Existential forgery:* succeeds in forging the signature of at least one message.
  - *Key only attack:* the adversary knows the public verification key but does not ask any queries.

RSA with a full domain hash function

- Signature is sig(m) = $(H(m))^d$ mod N.
  - $H()$ is such that its range is $[1,N]$

- *The system is no longer homomorphic*
  - *sig(m) · sig(m') ≠ sig(m·m')*

- *Seems hard to generate a random signature*
  - Computing s^e is insufficient, since it is also required to show m s.t. $H(m) = s^e$.

- Proof of security in the random oracle model – where $H()$ is modeled as a random function

The random oracle model

- In the real world, an attacker has access to the actual code that implements a hash function H.
- In our analysis attacker has only "oracle access" to H.
  - Attacker sends input x.
  - If this is the first query with this value, receives random H(x).
  - Otherwise, receives the value previously given for H(x).
- Proof strategy:
  - If there exists an attacker A that breaks a cryptosystem with random oracle access, then there exists an attacker B that contradicts the RSA assumption.
  - Namely, if we believe in the RSA assumption, then if we use a random oracle like hash function then the system is secure.

RSA with full domain hash –proof of security

- Claim: Assume that *H()* is a random function, then if there is a polynomial-time *A()* which performs existential forgery with non-negligible probability, then it is possible to invert the RSA function, on a random input, with non-negligible probability.

- Proof:
  - Our input: y. Our challenge is to compute y^d mod N.
  - Claim: *A()* which forges a signature of m, must query *H(m)*
  - *A()* queries *H()* and a signature oracle *sig()* (which computes the RSA function) and generates a signature *s* of a message for which it did not query *sig()*.
  - Suppose *A()* made at most *t* queries to *H()*, asking for $H(m_1),\ldots,H(m_t)$. Suppose also that it always queries *H(m)* before querying *sig(H(m))*.
  - We will show how to use *A()* to compute y^d mod N.

RSA with full domain hash –proof of security

- Proof (contd.)
- Let us first assume that *A* always forges the signature of m_t (the last query it sends to *H()*),
  - We can decide how to answer *A's* queries to *H(),sig()*.
  - Answer queries to *H()* as follows:
    - The answer to the t^{th} query (m_t) is *y*.
    - The answer to the j^{th} query ($j<t$) is $(r_j)^e$, where r_j is random.
  - Answer to *sig(x)* queries:
    - These are only asked for $x=H(m_j)$ where $j<t$. Answer with r_j. (Indeed $sig(H(m_j))= (H(m_j))^d = r_j$)
  - *A*'s output is (m_t,s).
    - If *s* is the correct signature, then we found y^d.
    - Otherwise we failed.
  - Success probability the same as the success probability of *A()*.

RSA with full domain hash –proof of security

- Proof (without assuming which m_i *A* will try to sign)
  - We can decide how to answer *A's* queries to *H(),sig()*.
  - Choose a random *i* in *[1,t]*, answer queries to *H()* as follows:
    - The answer to the *i*th query (m_i) is *y*.
    - The answer to the *j*th query ($j \neq i$) is $(r_j)^e$, where r_j is random.
  - Answer to *sig(x)* queries:
    - If $x=H(m_j)$, $j \neq i$, then answer with r_j. *Indeed* $sig(H(m_j))=(H(m_j))^d=r_j$
    - If $m=m_i$ then stop. (we failed)
  - *A*'s output is *(m,s).*
    - *If* $m=m_i$ and *s* is the correct signature, then we found y^d.
    - Otherwise we failed.
  - Success probability is *1/t* times the success probability of *A()*.

El Gamal signature scheme

- Invented by same person but different than the encryption scheme. (think why)
- A randomized signature: same message can have different signatures.
- Based on the hardness of extracting discrete logs
- The DSA (Digital Signature Algorithm/Standard) that was adopted by NIST in 1994 is a variation of El-Gamal signatures.

El Gamal signatures

- Key generation:
  - Work in a group Z_p^* where discrete log is hard.
  - Let g be a generator of Z_p^*.
  - Private key $1 < a < p\text{-}1$.
  - Public key $p, g, y=g^a$.

- Signature: (of M)
  - Pick random $1 < k < p\text{-}1$, s.t. gcd$(k,p\text{-}1)=1$.
  - Compute $m=H(M)$.
    - $r = g^k \bmod p$.
    - $s = (m - r \cdot a) \cdot k^{-1} \bmod (p\text{-}1)$
  - Signature is r, s.

El Gamal signatures

- Signature:
  - Pick random $1 < k < p\text{-}1$, s.t. gcd$(k,p\text{-}1)=1$.
  - Compute
    - $r = g^k$ mod p.
    - $s = (m - r \cdot a) \cdot k^{-1}$ mod $(p\text{-}1)$
- Verification:
  - Accept if
    - $0 < r < p$
    - $y^r \cdot r^s = g^m$ mod p

same *r* in both places!

- It works since $y^r \cdot r^s = (g^a)^r \cdot (g^k)^s = g^{ar} \cdot g^{m-ra} = g^m$
- Overhead:
  - Signature: one (offline) exp.   Verification: three exps.

El Gamal signature: comments

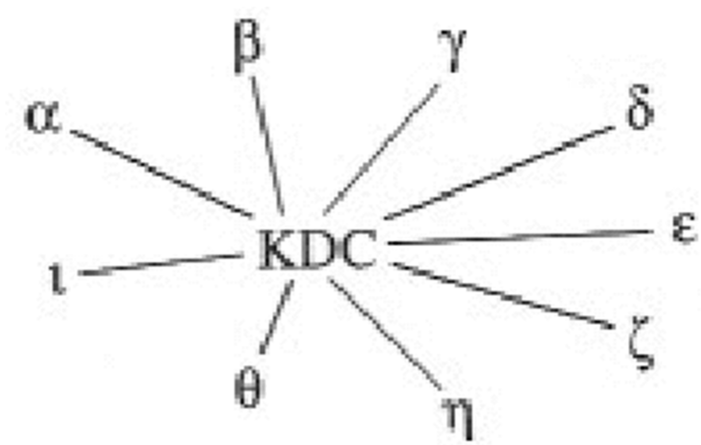
- Can work in any finite Abelian group
  - The discrete log problem appears to be harder in elliptic curves over finite fields than in Z_p^* of the same size.
  - Therefore can use smaller groups $\Rightarrow$ shorter signatures.
- Forging: find $y^r \cdot r^s = g^m \bmod p$
  - E.g., choose random $r = g^k$ and either solve dlog of g^m/y^r to the base r, or find $s=k^{-1}(m - \log_g y \cdot r)$ *(????)*
- Notes:
  - A different *k* must be used for every signature
  - If no hash function is used (i.e. sign *M* rather than *m=H(M)*), existential forgery is possible
  - If receiver doesn't check that $0<r<p$, adversary can sign messages of his choice.

Key Infrastructure for symmetric key encryption

- Each user has a shared key with each other user
  - A total of n(n-1)/2 keys
  - Each user stores n-1 keys

Key Distribution Center (KDC)

- The KDC shares a symmetric key K_u with every user *u*
- Using this key they can establish a trusted channel
- When *u* wants to communicate with *v*
  - *u* sends a request to the KDC
  - The KDC
    - *authenticates u*
    - *g*enerates a key K_{uv} to be used by *u* and *v*
    - *sends* $Enc(K_u, K_{uv})$ to *u*, and $Enc(K_v, K_{uv})$ to *v*



Key Distribution Center (KDC)

- Advantages:
  - A total of *n* keys, one key per user.
  - easier management of joining and leaving users.

- Disadvantages:
  - The KDC can impersonate anyone
  - The KDC is a single point of failure, for both
    - security
    - quality of service
- Multiple copies of the KDC
  - More security risks
  - But better availability

Trusting public keys

- Public key technology requires every user to remember its private key, and to have access to other users' public keys
- How can the user verify that a public key PK_v corresponds to user *v*?
  - What can go wrong otherwise?

- A simple solution:
  - A trusted public repository of public keys and corresponding identities
    - Doesn't scale up
    - Requires online access per usage of a new public key

Certification Authorities (CA)

- A method to bootstrap trust
  - Start by trusting a single party and knowing its public key
  - Use this to establish trust with other parties (and associate them with public keys)

- The Certificate Authority (CA) is trusted party.
  - All users have a copy of the public key of the CA
  - The CA signs Alice's digital certificate. A simplified certificate is of the form *(Alice, Alice's public key)*.

Certification Authorities (CA)

- When we get Alice's certificate, we
  - Examine the identity in the certificate
  - Verify the signature
  - Use the public key given in the certificate to
    - Encrypt messages to Alice
    - Or, verify signatures of Alice

- The certificate can be sent by Alice without any online interaction with the CA.

Certification Authorities (CA)

- Unlike KDCs, the CA does not have to be online to provide keys to users
  - It can therefore be better secured than a KDC
  - The CA does not have to be available all the time
- Users only keep a single public key – of the CA
- The certificates are not secret. They can be stored in a public place.
- When a user wants to communicate with Alice, it can get her certificate from either her, the CA, or a public repository.
- A compromised CA
  - can mount active attacks (certifying keys as being Alice's)
  - but it cannot decrypt conversations.

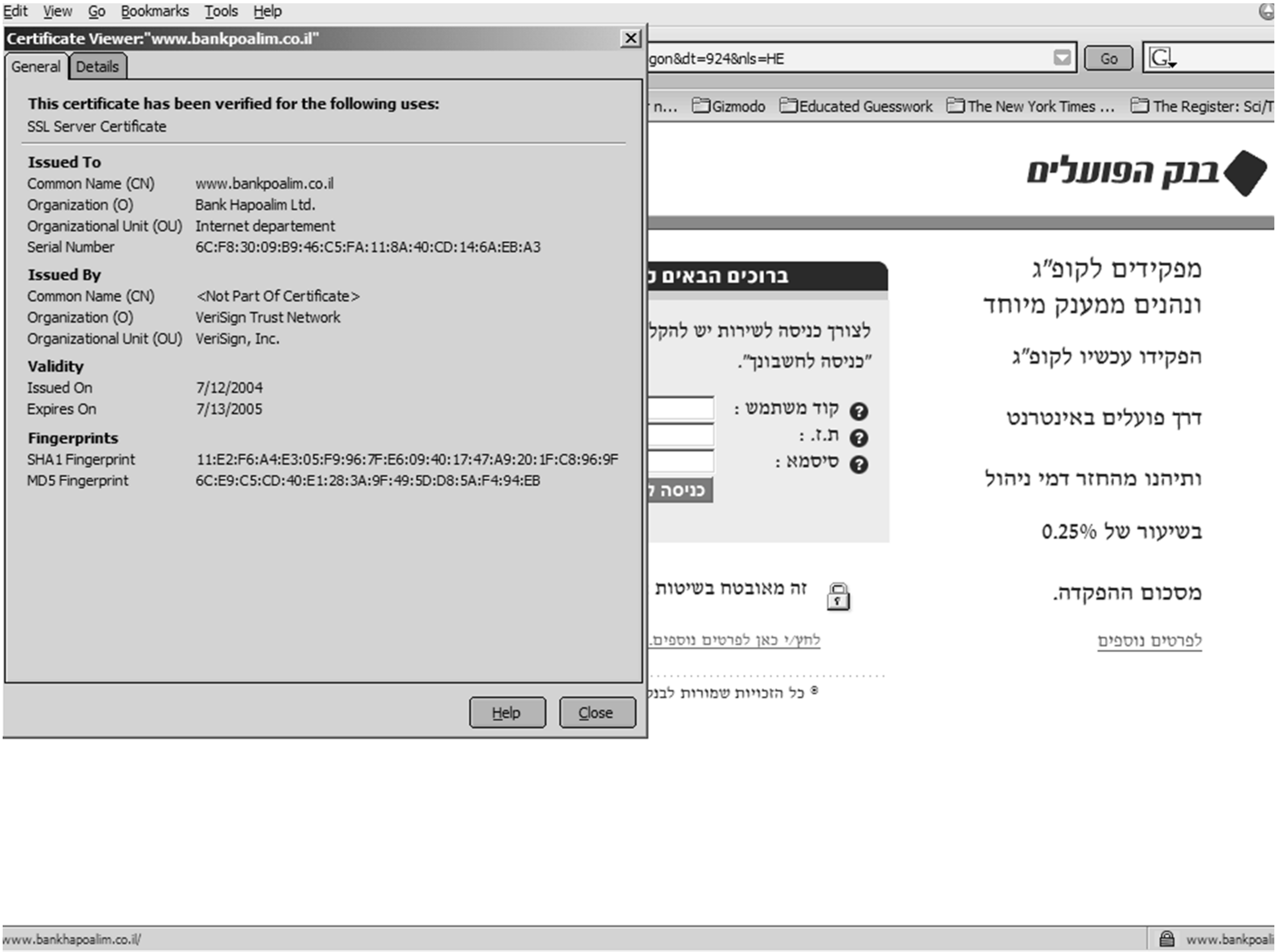
Certification Authorities (CA)

- An example.
  - To connect to a secure web site using SSL or TLS, we send an https:// command
  - The web site sends back a public key[1], and a certificate.
  - Our browser
    - Checks that the certificate belongs to the url we're visiting
    - Checks the expiration date
    - Checks that the certificate is signed by a CA whose public key is known to the browser
    - Checks the signature
    - If everything is fine, it chooses a session key and sends it to the server encrypted with RSA using the server's public key

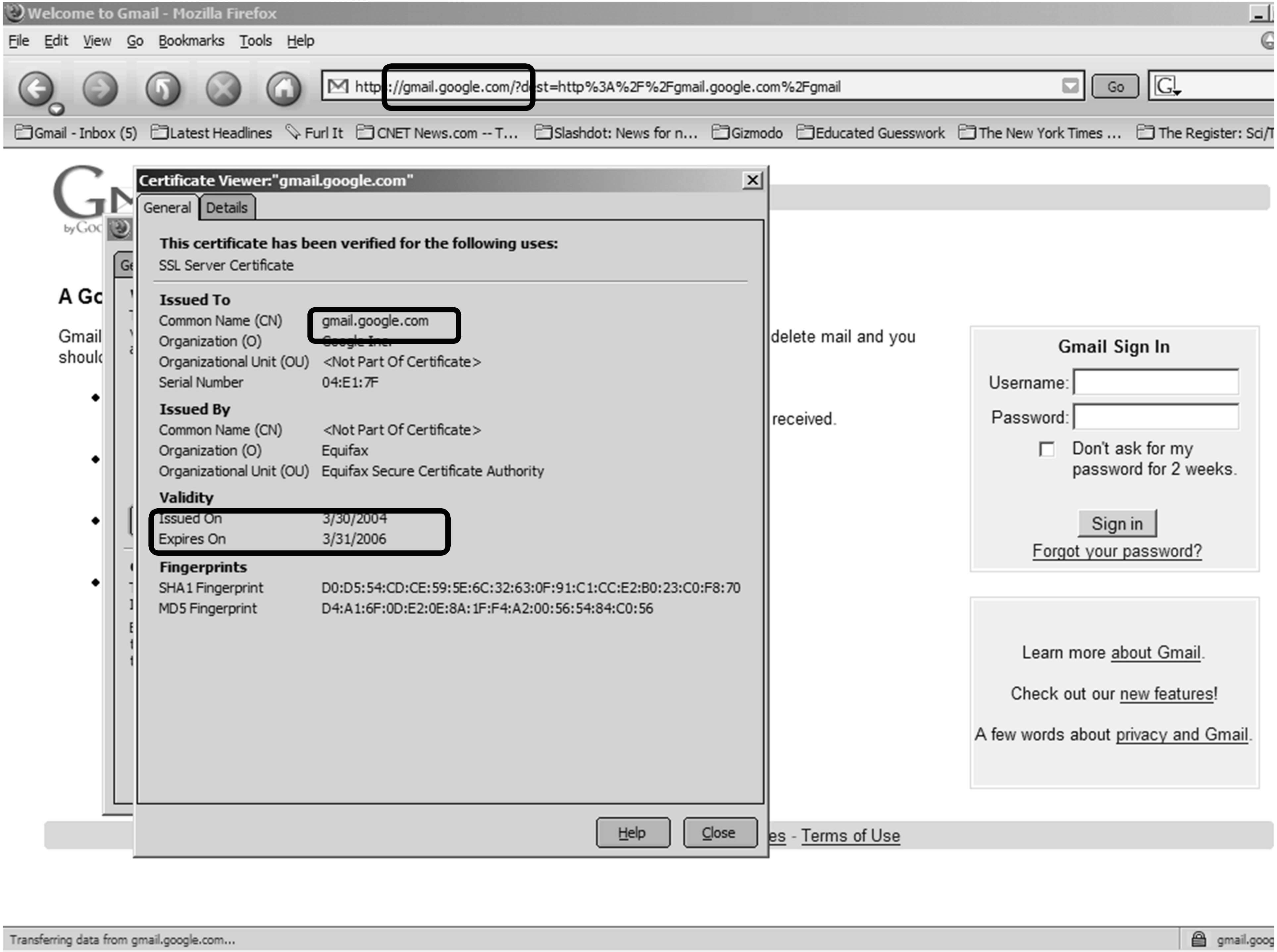
(1) This is a very simplified version of the actual protocol.

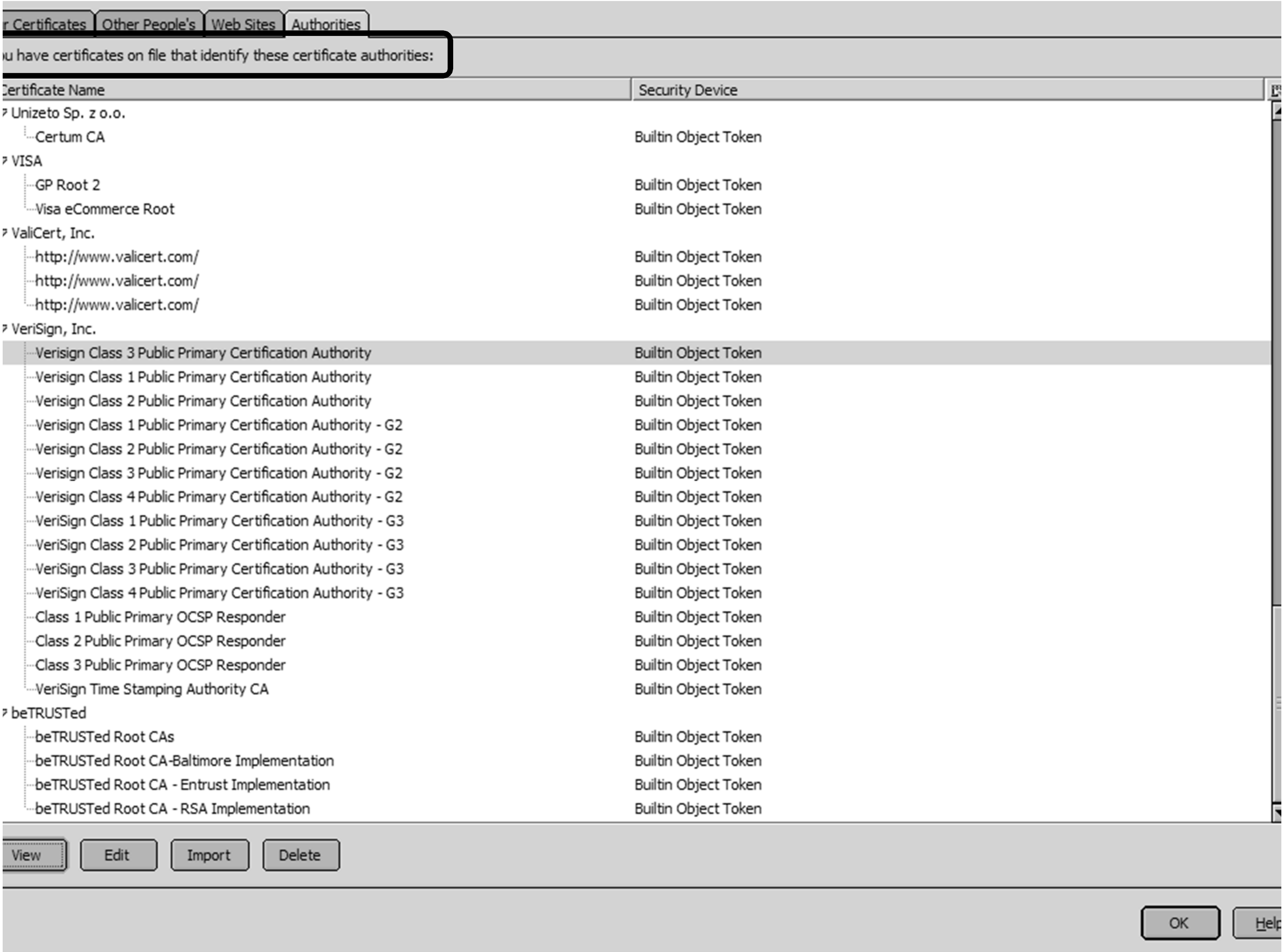
An example of an X.509 certificate

```
Certificate:
  Data:
    Version: 1 (0x0)
    Serial Number: 7829 (0x1e95)
    Signature Algorithm: md5WithRSAEncryption
    Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
      OU=Certification Services Division, CN=Thawte Server
      CA/emailAddress=server-certs@thawte.com
    Validity
         Not Before: Jul 9 16:04:02 1998 GMT
         Not After : Jul 9 16:04:02 1999 GMT
    Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala, OU=FreeSoft,
      CN=www.freesoft.org/emailAddress=baccala@freesoft.org
    Subject Public Key Info:
         Public Key Algorithm: rsaEncryption
         RSA Public Key: (1024 bit)
         Modulus (1024 bit): 00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
           33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
           66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
           70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
           16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
           c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
           8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
           d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8: e8:35:1c:9e:27:52:7e:41:8f
         Exponent: 65537 (0x10001)
  Signature Algorithm: md5WithRSAEncryption
    93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
      92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:…
```


Edit View Go Bookmarks Tools Help
Certificate Viewer:"www.bankpoalim.co.il"
General Details
This certificate has been verified for the following uses:
SSL Server Certificate
Issued To
Common Name (CN) www.bankpoalim.co.il
Organization (O) Bank Hapoalim Ltd.
Organizational Unit (OU) Internet departement
Serial Number 6C:F8:30:09:B9:46:C5:FA:11:8A:40:CD:14:6A:EB:A3
Issued By
Common Name (CN) <Not Part Of Certificate>
Organization (O) VeriSign Trust Network
Organizational Unit (OU) VeriSign, Inc.
Validity
Issued On 7/12/2004
Expires On 7/13/2005
Fingerprints
SHA1 Fingerprint 11:E2:F6:A4:E3:05:F9:96:7F:E6:09:40:17:47:A9:20:1F:C8:96:9F
MD5 Fingerprint 6C:E9:C5:CD:40:E1:28:3A:9F:49:5D:D8:5A:F4:94:EB
Help Close
gon&dt=924&nls=HE
Go
Gizmodo Educated Guesswork The New York Times ... The Register: Sci/T
בנק הפועלים
מפקידים לקופ"ג
ונהנים ממענק מיוחד
הפקידו עכשיו לקופ"ג
דרך פועלים באינטרנט
ותיהנו מהחזר דמי ניהול
בשיעור של 0.25%
מסכום ההפקדה.
לפרטים נוספים
קוד משתמש :
ת.ז. :
סיסמא :
www.bankhapoalim.co.il/
www.bankpoali

[redacted]


Welcome to Gmail - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://gmail.google.com/?dest=http%3A%2F%2Fgmail.google.com%2Fgmail
Go
Gmail - Inbox (5) Latest Headlines Furl It CNET News.com -- T... Slashdot: News for n... Gizmodo Educated Guesswork The New York Times ... The Register: Sci/T
Certificate Viewer:"gmail.google.com"
General Details
This certificate has been verified for the following uses:
SSL Server Certificate
Issued To
Common Name (CN) gmail.google.com
Organization (O) Google Inc.
Organizational Unit (OU) <Not Part Of Certificate>
Serial Number 04:E1:7F
Issued By
Common Name (CN) <Not Part Of Certificate>
Organization (O) Equifax
Organizational Unit (OU) Equifax Secure Certificate Authority
Validity
Issued On 3/30/2004
Expires On 3/31/2006
Fingerprints
SHA1 Fingerprint D0:D5:54:CD:CE:59:5E:6C:32:63:0F:91:C1:CC:E2:B0:23:C0:F8:70
MD5 Fingerprint D4:A1:6F:0D:E2:0E:8A:1F:F4:A2:00:56:54:84:C0:56
Help Close
delete mail and you
received.
Gmail Sign In
Username:
Password:
Don't ask for my password for 2 weeks.
Sign in
Forgot your password?
Learn more about Gmail.
Check out our new features!
A few words about privacy and Gmail.
Terms of Use
Transferring data from gmail.google.com...


r Certificates
Other People's
Web Sites
Authorities
u have certificates on file that identify these certificate authorities:
Certificate Name
Security Device
Unizeto Sp. z o.o.
Certum CA Builtin Object Token
VISA
GP Root 2 Builtin Object Token
Visa eCommerce Root Builtin Object Token
ValiCert, Inc.
http://www.valicert.com/ Builtin Object Token
http://www.valicert.com/ Builtin Object Token
http://www.valicert.com/ Builtin Object Token
VeriSign, Inc.
Verisign Class 3 Public Primary Certification Authority Builtin Object Token
Verisign Class 1 Public Primary Certification Authority Builtin Object Token
Verisign Class 2 Public Primary Certification Authority Builtin Object Token
Verisign Class 1 Public Primary Certification Authority - G2 Builtin Object Token
Verisign Class 2 Public Primary Certification Authority - G2 Builtin Object Token
Verisign Class 3 Public Primary Certification Authority - G2 Builtin Object Token
Verisign Class 4 Public Primary Certification Authority - G2 Builtin Object Token
VeriSign Class 1 Public Primary Certification Authority - G3 Builtin Object Token
VeriSign Class 2 Public Primary Certification Authority - G3 Builtin Object Token
VeriSign Class 3 Public Primary Certification Authority - G3 Builtin Object Token
VeriSign Class 4 Public Primary Certification Authority - G3 Builtin Object Token
Class 1 Public Primary OCSP Responder Builtin Object Token
Class 2 Public Primary OCSP Responder Builtin Object Token
Class 3 Public Primary OCSP Responder Builtin Object Token
VeriSign Time Stamping Authority CA Builtin Object Token
beTRUSTed
beTRUSTed Root CAs Builtin Object Token
beTRUSTed Root CA-Baltimore Implementation Builtin Object Token
beTRUSTed Root CA - Entrust Implementation Builtin Object Token
beTRUSTed Root CA - RSA Implementation Builtin Object Token
View
Edit
Import
Delete
OK
Help

Certificates

- A certificate usually contains the following information
  - Owner's name
  - Owner's public key
  - Encryption/signature algorithm
  - Name of the CA
  - Serial number of the certificate
  - Expiry date of the certificate
  - …
- Your web browser contains the public keys of some CAs
- A web site identifies itself by presenting a certificate which is signed by a chain starting at one of these CAs

An example of an X.509 certificate

```
Certificate:
  Data:
    Version: 1 (0x0)
    Serial Number: 7829 (0x1e95)
    Signature Algorithm: md5WithRSAEncryption
    Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
      OU=Certification Services Division, CN=Thawte Server
      CA/emailAddress=server-certs@thawte.com
    Validity
         Not Before: Jul 9 16:04:02 1998 GMT
         Not After : Jul 9 16:04:02 1999 GMT
    Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala, OU=FreeSoft,
      CN=www.freesoft.org/emailAddress=baccala@freesoft.org
    Subject Public Key Info:
         Public Key Algorithm: rsaEncryption
         RSA Public Key: (1024 bit)
         Modulus (1024 bit): 00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
           33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
           66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
           70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
           16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
           c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
           8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
           d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8: e8:35:1c:9e:27:52:7e:41:8f
         Exponent: 65537 (0x10001)
  Signature Algorithm: md5WithRSAEncryption
    93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
      92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:…
```

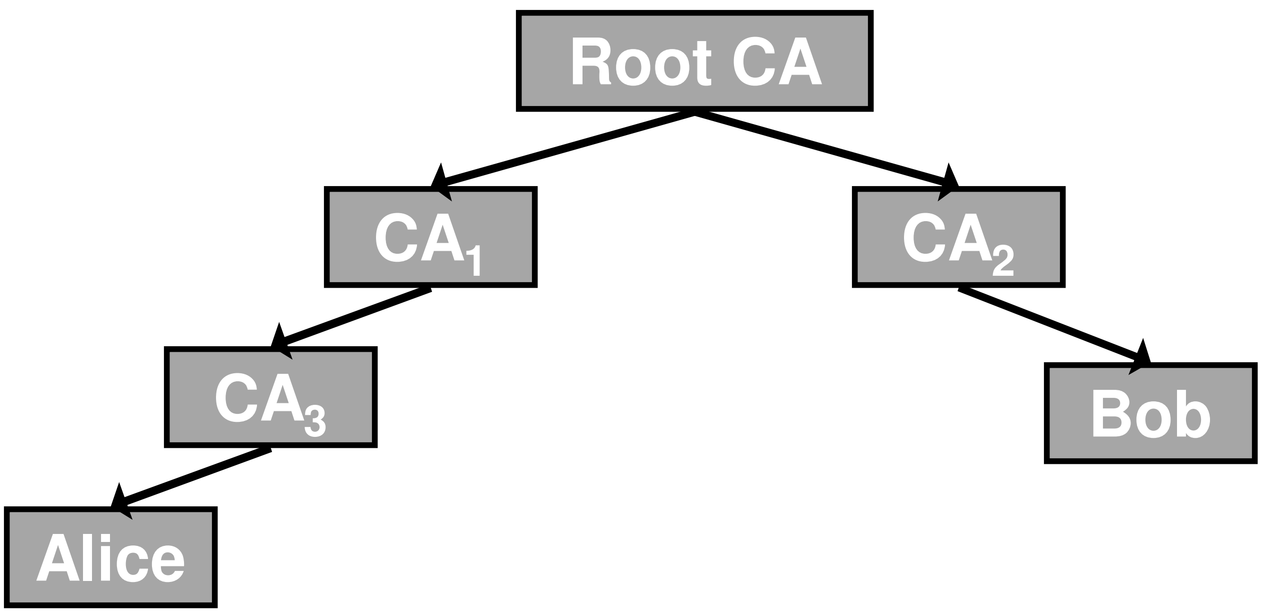
Public Key Infrastructure (PKI)

- The goal: build trust on a global level

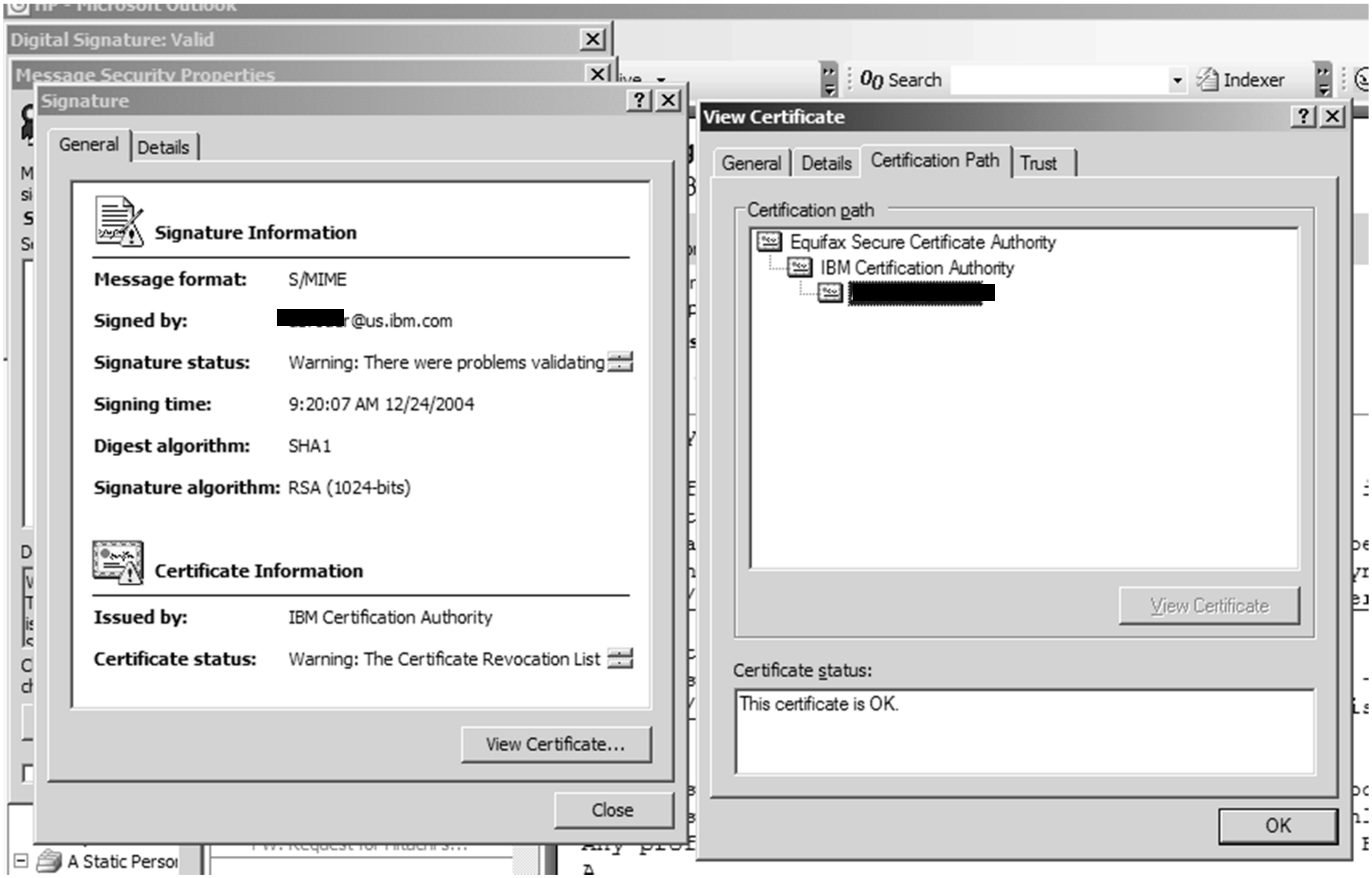
- Running a CA:
  - If people trust you to vouch for other parties, everyone needs you.
  - A license to print money
  - But,
    - The CA should limit its responsibilities, buy insurance…
    - It should maintain a high level of security
    - Bootstrapping: how would everyone get the CA’s public key?

Public Key Infrastructure (PKI)

- Monopoly: a single CA vouches for all public keys
  - Mostly suitable for enterprises.
- Monopoly + delegated CAs:
  - top level CA can issue *special* certificates for other CAs
  - Certificates of the form
    - [$(\text{Alice}, PK_A)_{CA3}$, $(\text{CA3}, PK_{CA3})_{CA1}$, $(\text{CA1}, PK_{CA1})_{\text{ROOT-CA}}$]



Certificate chain



Revocation

- Revocation is a key component of PKI
  - Each certificate has an expiry date
  - But certificates might get stolen, employees might leave companies, etc.
  - Certificates might therefore need to be revoked before their expiry date
  - New problem: before using a certificate we must verify that it has not been revoked
    - Often the most costly aspect of running a large scale public key infrastructure (PKI)
    - How can this be done efficiently?