Hybrid Cryptoschemes and Public Key Authentication

Alessandro Barenghi

Dipartimento di Elettronica, Informazione e Bioingegneria (DEI)
Politecnico di Milano

alessandro.barenghi - at - polimi.it

Overview

Lesson contents

- Hybrid Cryptosystems
- Digital Certificates
- The Web of Trust model

Communication security

The trivial solution

- Alice and Bob want to have a confidential, authenticated communication.
- Trivial solution: they **meet in person**, and exchange a secret key k
- From the meeting point on, they use a **symmetric cipher** to encrypt their communications
- Confidentiality is achieved, authentication is implicit from the secrecy of the symmetric key
- This was the only solution possible up to 1976, as only symmetric ciphers were available

Issues

Does it scale?

- **Problem 1**: To communicate with n people, Alice needs to know/store n separate keys: **key storage is an issue**
- **Problem 2**: To communicate with someone, Alice needs to meet him/her **in person once**, or have **a secure communication channel**
- **Solution**: Alice generates an asymmetric keypair k^A_{pub}, k^A_{pri} and makes k^A_{pub} available, and so does Bob generating k^B_{pub}, k^B_{pri}
- To communicate with Bob, Alice **signs** the message with k^A_{pri}, and **encrypts** it with k^B_{pub}

Efficiency issues

Hybrid cryptosystems

- Asymmetric ciphers are slow ($\approx$ 100kiB/s on a desktop): **encryption** and **signature** of large messages are **unpractical**
- Build a **hybrid cryptosystem** using multiple cryptographic primitives
  - Encryption: Alice generates a random symmetric key k, encrypts the message with it, and encrypts k with k^B_{pub}
  - Decryption: Bob decrypts k using k^B_{pri}, and uses it to decrypt the message, after which he may discards k.
  - Signature: Alice should sign something **smaller than the whole message**, although **uniquely bound** to it in some way
- Solving the efficient signature problem requires a new cryptographic primitive, the cryptographic hash

Cryptographic Hash

A non-encrypting cryptographic primitive

- A hash is a deterministic function from **arbitrary** length message m, to **fixed length** output $h = H(m)$
- Practically used to obtain a fixed-length "label" h for a digital object
- The same value of h may be the hash of different messages (no such thing as bijective arbitrary size $\rightarrow$ fixed size compression)
- For a cryptographic hash it should not be **practically feasible** to:
  1. Given h, find a m such that $H(m) = h$ (1st preimage)
  2. Given a message m, find m' such that $H(m') = H(m)$ (2nd preimage)
  3. Find two messages m', m'' such that $H(m') = H(m'')$ (collision)
- Note that 1st preimage $\Rightarrow$ 2nd preimage $\Rightarrow$ collision,

One last(ing) issue

Public key authentication

- Hybrid cryptoschemes allow efficient, confidential communication
- Crucial assumption: Alice and Bob are employing **each other's** public key to send the messages
- If no measures are taken, it is easy to perform an impersonation attack on public key cryptosystems
- We need some form of **digital certificate** of authenticity of public keys
- We will see solutions to this issue, and the infrastructures built to manage their lifecycle (certificate issuing-use-revocation-expiration)

Digital certificates

What are digital certificates?

- Digital certificates are designed mimicking an actual paper certificate for the information we want to authenticate
- Our purpose is to **authenticate the binding** of a public **key** to the **identity** of someone/some company
- The format for standardised digital certificates is defined through a formal language through Abstract Syntax Notation One (ASN.1)
- The contents of a digital certificate and their semantic values are specified employing ASN.1 in the X.509 Standard by ITU-T (International Telecommunication Union)
- The certificate signatures are performed employing signing the output of a cryptographic hash with an asymmetric primitive

Digital certificates

Format

- The digital certificates are commonly exchanged as files
- Most common encoding: Distinguished Encoding Rules (DER, ITU X.690) specifies a binary encoding format for the certificate
- DER encoding includes non printable characters,
  - The DER encoded certificate may be encoded in Base64 yielding data that can be copy-pasted easily (common file extension: PEM)
- Parsing certificates should be done carefully: string fields may contain also non-printable characters (strings stored as length+content)

Digital certificates

Contents

- The X.509 standard **mandates** the certificate for the digital signature **to contain** the following information:
  - **Version and Serial Number** : Used identify the certificate and specify the accepted extensions
  - **Subject**: The person, or entity to which the key belongs.
  - **Issuer**: The entity that verified the information and issued the certificate.
  - **Valid-From/Valid-To**: The validity period of the certificate.
  - **Key-Usage**: Purpose of the public key (e.g. encryption, signature verification on data, signature verification on certificates...).
  - **Public Key**: The public key to be bound to the subject.
  - **Signature Algorithm**: The algorithm used to sign the hash of the certificate and the hash algorithm.
  - **Signature**: The actual signature of the hash of the certificate.

Digital certificates

Certificate inspection

- Inspecting X.509 certificates manually requires a pretty-printer
- Modern browsers include a summary pretty printer
- The openssl library client performs prettyprinting if used as: `openssl x509 -inform DER -in <cert.der> -noout -text`
- A decoder for DER files can be found at `http://lapo.it/asn1js/`

Digital certificates

How is it signed?

- The signature is performed through hashing part of the certificate and signing the digest as per RFC 5280.
- The parts of the certificate which must be signed include:
  - Version and Serial Number
  - Subject, Issuer and Validity dates
  - The public key to be certified
  - Some of the certificate extensions (e.g. key usage)
- The signature process takes as input the aforementioned fields, encoded in DER in order, hashes them and signs the hash with k_{pri}^{Issuer}
- RFC 5280 specifies that either OS2IP (a custom sequence of bytewise additions!) or PKCS1 standard hashing must be used
- In practice, **PKCS1** is the most common choice since it mandates the use of a **cryptographically strong hash**

Digital certificates

Possible issues

- The signature is reliable iff the employed hash is **collision**-resistant and the signature algorithm is not broken
- One of the most common choices for the hash was $\mathrm{MD5}$
- Today, computing a collision for $\mathrm{MD5}$ takes $\approx$ 10s on this laptop
- Given a message m_1, you can derive a colliding message is in the form $m_2 = m_1 || \mathrm{pad}$, where pad is a properly chosen 64 B string
- Certificates signed with $\mathrm{MD5}$ can be forged taking a valid certificate, changing the desired values, and stuffing the pad as the comment field in the certificate extensions (last part to be fed to $\mathrm{MD5}$)
- More of this to come in the lesson on cryptographic hashes

Practical key sizes

A brief primer on security margins

- A proper security margin is ensured choosing a large enough keyspace for all the ciphers involved
- The table obtained from the estimates at `www.keylength.com` by both academia and national standard bodies

Safe up to year	Secret/Private Key length [b] RSA	ECDSA	DSA (key/group)	AES Equiv.
1986	512	111	122/515	60
2010	1024	144	144/1024	80
2030	2048	224	224/2048	112
2060	4440	256	256/4440	128
Fores. Future	26268	512	512/26268	256

Digital certificates

Who signs the certificates?

- “Someone trusted” signs the certificate, vouching for its authenticity
- To provide this trusted entity, three solutions have been proposed:
  - PKI : **Public Key Infrastructure**: A **centralized, tree structured** architecture of entities which sign certificates of their subsiders. The root authorities are implicitly trusted
  - WoT : **Web-of-Trust**: A **distributed** architecture relying on the “small world assumption” where everyone can sign certificates. The trust on the authenticity of a certificate is established depending on the trust on the authenticity of its signers

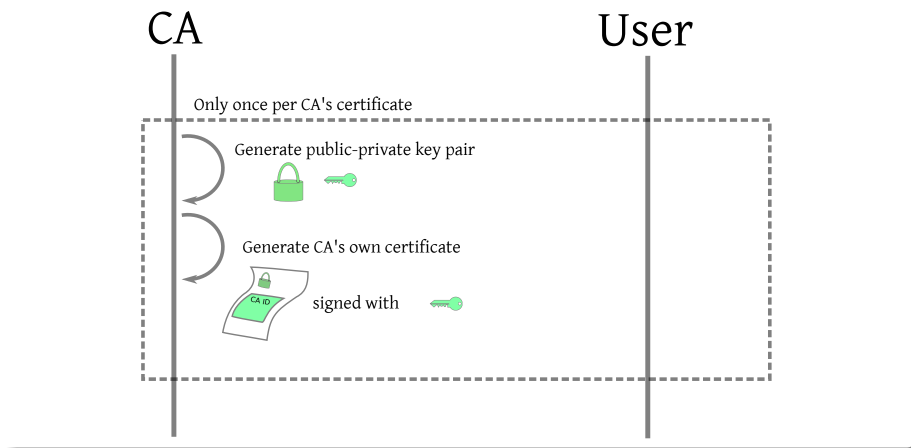
PKI Infrastructure

Roles

- **Certification Authority (CA)**: An entity (typically a firm) which takes care of signing the certificates, and distributes its public key to all the users in a trusted manner
- **Registration Authority (RA)**: An entity (again, usually a firm) which takes care of verifying the actual authenticity of a certificate, gathering data on the user (physically checking his/her identity and certificate hash). Very often, it coincides with the CA
- **User**: Asks the CA to sign his certificate, or employs the CA public keys to verify the authenticity of the certificates for another user.

PKI Actions

Bootstrapping a CA



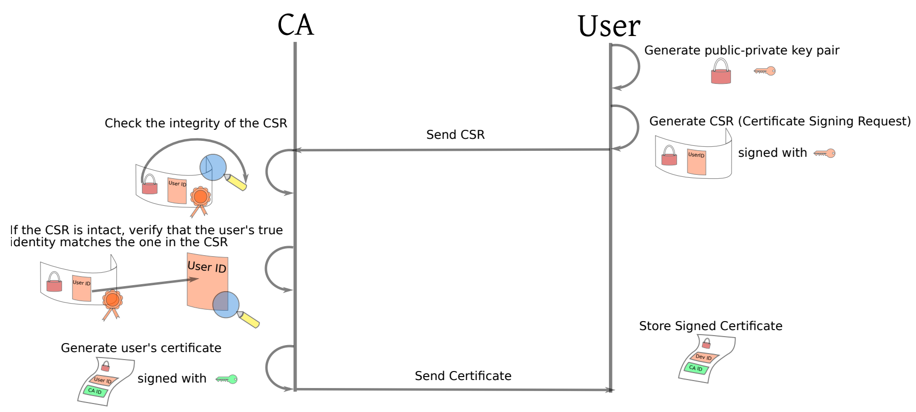
PKI Actions

Bootstrapping a CA

- The CA creates its own keypair and fills in the fields of its certificate
- We assume, for the sake of clarity, that we are describing a root CA, i.e. one sitting at the top of the PKI hierarchy
- The CA self-signs its own certificate, and stores its own private key in a tightly guarded place (at least,hopefully)
- The CA takes care to distribute its self-signed public key certificate in a safe manner to the largest possible user-base
- Typical ways to do so include:
  - Embedding the certificates in applications, e.g., web browsers
  - Embedding the certificates within operating system distribution chains

PKI Actions

Issuing a Certificate



PKI Actions

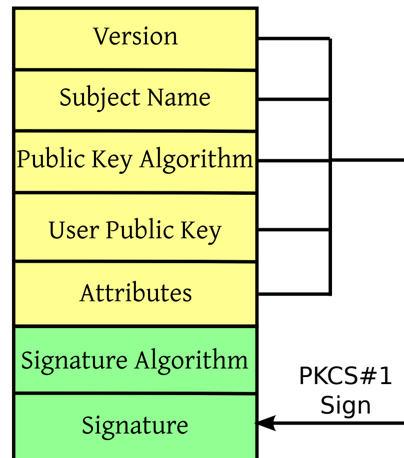
Issuing a Certificate

- The user willing to get his public key certified starts by generating safely a keypair
- The certificate signature request CSR is PKCS10 standardised and self-signed by the user
- PKCS10 mandates the use of X.509 format to represent the CSR
- The CA should verify either via a RA or directly whether the actual user/firm who generated the certificate is the one who claims to be
- Once the certificate is signed, it can be publicly used and does not need to be sent over a secure channel to the user
- The user can check that the CA signature is authentic

PKI Actions

Issuing a certificate - CSR Structure (specified in the X.509 stardard)

- The user has generated a keypair
- The user performs the PKCS signature as usual
  - The yellow part of the CSR is DER encoded
  - The yellow part of the CSR is hashed
  - The hash of the yellow part is signed **with the user's private key** and the signature is stored



PKI Actions

Practical CSR creation

- A private key can be created with the following openssl command

```
openssl genrsa -out example_client.key <key_size>
```

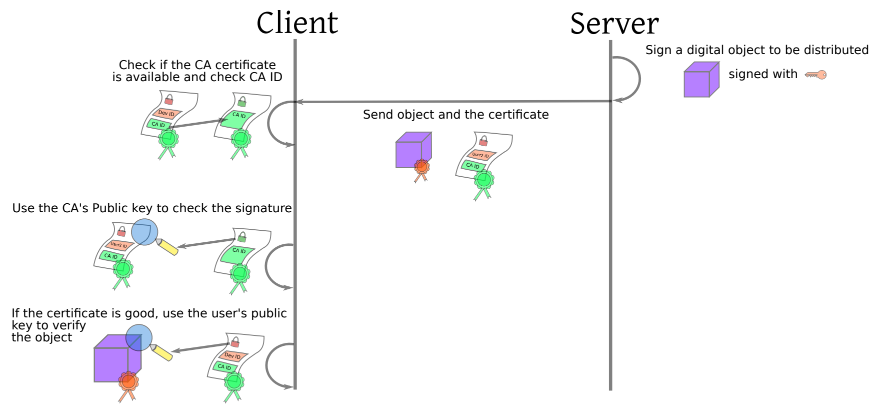
- The CSR creation automatically generates the public key

```
openssl req -new -key example_client.key -out client.csr
```

- The user will deliver the CSR to the CA in any preferred way
- The CA will return a complete X.509 certificate to the user
- If you need to setup a CA, you'll find the Easy-RSA set of scripts handy `https://github.com/OpenVPN/easy-rsa`

PKI Actions

Checking the authenticity of a digital object



PKI Actions

Checking the authenticity of a digital object

1. The client receives an object from a server together with a certificate of the public key to verify the signature
2. The client checks whether the CA of the certificate is among the ones he trusts (i.e. has an authentic public key for it)
3. The client checks that the server-supplied certificate is authentic, i.e. verifies the signature made by the CA on the server public key
4. If the certificate is authentic, the client verifies the signature on the object with the public key contained in the certificate
5. If the signature of the object is correct, the object is authentic, if **any** single one of the previous step fails it is not

PKI Actions

Compromised certificates: how to deal with them

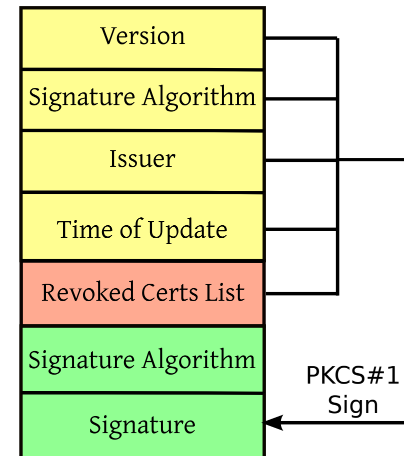
- A keypair can be compromised (e.g. by theft, blackmail)
- A way to invalidate the certificate emitted by a CA is needed to prevent signatures on forged objects
- This action, namely the **revocation** of a certificate can be notified via:

CRLs: **Certificate Revocation Lists** are lists of certificate IDs containing all the compromised certificates of the CA, available for download

OCSP: **Online Certificate Status Protocol** is an online protocol which allows a user to query the CA which has signed a certificate to ascertain its revocation state.

PKI Actions

CRL Method - Format



- The revoked certificates list is a sequence of
  - Revoked certificate identifier
  - Date of revoking
  - Optional extensions
- The CRL may optionally contain the next update date
- URL where the CRL is distributed contained in the CA certificate

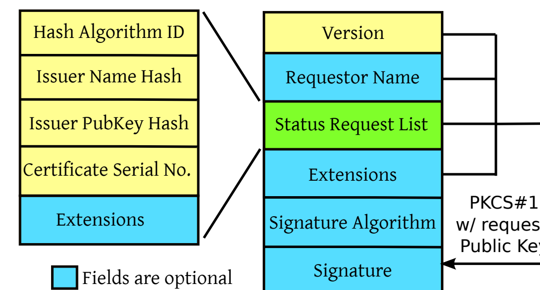
PKI Actions

OCSP Method

- The size of the CRLs increased, and due to the fact that they may not be updated in realtime when certificates are revoked, the Online Certificate Status Protocol was developed (RFC 2560).
- The OCSP verification of the status of a certificate takes place as an online request-response between the user and the signer CA
- The user queries the CA for the status of the certificate, and the CA answers with a message notifying the current state of the certificate
- The information is fresher, at the cost of mantaining a relatively small sized database at the CA's side, which is no longer an issue
- To avoid bandwidth waste and latencies, usually a local cache of the most recent OCSP transactions is mantained by the user

PKI Actions

OCSP Method - OCSP Request and Response Formats



- The OCSP request is only optionally signed by the requester
- The hashes are performed from the DER encoding of the names
- RFC2459 mandates the inclusion of a nonce in the extensions [X.509 v3 only]

PKI Hierarchical structure

Hierarchical structure

- Up to now, we have assumed that the CA was the ultimate authority for warranting the authenticity of its certificates
- As the PKI system was designed to scale up to a large number of users a hierarchy among CA was established
- A CA certificates the public key of the ones one level below it, and has its public key certified by the one which is one level above it
- This effectively builds a tree-based structure where the leaves are the users and the root is represented by the so-called **root CA**s.

PKI Hierarchical structure

Hierarchical authenticity verification

- The differences from a flat CA structure are the following ones:
  - **Issuing**: Certificates are issued to the users only by the leaf CA
  - **Revocation**: Certificates are revoked by the CA which emitted them
  - **Verification**: The verification step is performed as follows:
    1. User checks if the certificate he wants to authenticate is signed by one of the CAs trusted by him (he has a copy of their cert.s)
    2. **Yes** (**Base case**): The authenticity is checked with the public key of the trusted CA. If the certificate is authentic, he goes back to what he was checking before.
    3. **No** (**Recursion step**): User obtains the certificate of the CA which signed the certificate currently under exam, and tries to authenticate it. Go to 1. When the user knows the certificate is authentic, he can check the one currently under exam and go back to what it was checking before.

PKI Actions

PKI Hierarchical structure - Common Pitfalls

- Root CAs are a **single point of failure** for the whole infrastructure: if a Root CA gets its key compromised, all the certificates issued from offspring CAs become forgeable.
- The scheme still needs to **distribute the certificates** of the Root CAs in a safe way and promptly remove Root CA compromised keys
- Since the chain of trust is indicated in the extensions field of the certificate, **all the extensions** should be properly **parsed**
- The signature keys employed should match or exceed the advised **key length** by NIST
- CAs should work at a equal- or higher- security level than the users

Automating Certificate management

The ACME protocol

- Asking for the issuance or revocation of a certificate requires procedures which have typically a human-in-the-loop
  - This may be effective for highly valuable assets (e.g. bank websites), but slows makes prompt certificate renewal/revocation slower
- The ACME Protocol[a] was designed to automate the request, renewal and revocation of certificates employing the ability of serving a given content over HTTPS as a proof-of-identity
- A fully working implementation is Certbot, which uses the *Let's encrypt* CA to provide 90-days lasting X.509 certificates

[a] Currently, an Internet-Draft
`https://tools.ietf.org/html/draft-ietf-acme-acme-06`

Web Of Trust

A distributed alternative

- The alternate model to provide authentication for unknown public key is the Web-of-Trust scheme
- In this scheme, invented for private mail communication by Phil Zimmermann, everyone may act as a CA
- An user may pick the one he favours to act as CAs: the key/identity bindings signed by them will be deemed authentic
- It is possible to extend the trust to trusted-users-by-trusted users
- The scheme works on the "small world assumption": there are ≈ 6 degrees of separation among two random person on the planet.

Web Of Trust

Actors

- **Users**: The only actor of the scheme are users, they :
  - Encrypt/sign/verify a digital object (file/mail message)
  - Acts as a CA signing someone else's key/id pair
  - Keep a local key/id *keyring* where all the certificates are kept
  - Choose which users they trust to be acting as a CA
- **Keyservers**: Provide a globally synchronized, trusted archive of public certificates.
  - Public lookup is provided via either key ID or user ID
  - The archive is append only, no practical removal is possible
  - Anyone can run a keyserver, provided he has enough resources.
  - A list of keyservers is available here : `http://www.pgpi.org/services/keys/keyservers/`
  - We run one at `pgp-srv.deib.polimi.it`

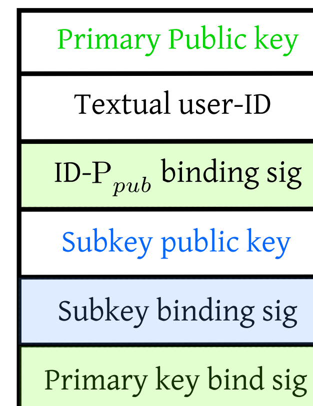
Web Of Trust

Certificate Format

- The key/identity pairs are **not** stored in a **X.509** compliant certificate, but rather in a format specified by RFC4880
- The format is a sequence of data blocks (*packets*) which contain:
  - A one-byte **identifier of the content** of the packet (identity, public key, signature, private key, object being signed)
  - A one to four bytes value specifying **the length** of the object (big endian byte ordering)
  - The actual packet **content**
- The format is binary but it is common to encode it in a printable form
- The encoding chosen is Radix-64, a variant of Base64 encompassing a mandatory, per line, CRC to allow the end user to check the integrity of the message (against accidental transmission errors)

Web Of Trust

Certificate contents and creation



- An OpenPGP certificate contains a primary public key and a textual ID
- Upon creation, the owner self-signs it and appends the signature
- Secondary public keys may be included (one-key-per-purpose principle)
- Subkeys are bound to the main key via both a self verifiable signature, and one verifiable with the public key
- Signatures from users acting as CAs are appended to the certificate

Web Of Trust

Certificate management and differences from classic PKI

- The typical lifecycle of a certificate in the web of trust model is:
  - The user **generates a keypair**, and stores it in his own keyring, self signing the public key
  - The user **exports a self-signed certificate** to the public keyservers
  - The user meets other users, and, after checking mutually their identities, they **cross-sign their public key certificates**
  - The newly signed **certificates are uploaded** on the public keyservers, which merge them with the old ones
- The key/identity verification before a certificate is signed is demanded to the users, assuming that they are careful in the process

Web Of Trust

Authenticity management

- A certificate is deemed to be authentic depending on the level of trust assigned to the CA-user signed it
- Trust levels for a CA-user are stored locally in the so called *trust-db*
- The default trust levels in the WoT scheme are :
  - **ultimate**: It's the user's own key trust level. Every key you sign with an "ultimate" trusted key becomes authentic.
  - **complete**: One signature by a key with this level of trust makes the key/id pair authentic.
  - **marginal**: At least 2 keys with marginal trust have to sign a key/id pair to make it authentic.
  - **untrusted**: Untrusted key signatures do not contribute to mark a key/id pair as authentic
- The aforementioned criteria are the default ones for GPG, the actual number of signatures/trust levels can be customized

Web Of Trust

Certificate Revocation

- Since anyone can be a CA, the revocations are issued on a per-signature basis
- To revoke a signature, a so-called revocation signature is performed and appended to the certificate. Typically :
  - The owner of the certificate may revoke the self signature on it
  - A CA-user may revoke its signature to the certificate individually: the other signatures will still be valid
- For the revocation to be public, the updated certificate should be uploaded on the keyservers (and downloaded by the clients)
- The OpenPGP specifications suggest that a revoked signature should carry almost the same weight of a revoked certificate
- **Issue**: what if I sign and then revoke the signature on someone else's certificate to discredit him?

Web Of Trust

Pitfalls and Issues

- **Direct verification** of the authenticity of a certificate should be performed in person in order **to bootstrap** the scheme
- **Prompt revocation** of compromised certificates is not easy if a user does not update regularly the keys in his keyring
- The certificates are identified by a unique ID: the original choice for the **ID length was insufficient** (32 b), thus it is possible to have collisions on certificate IDs (this has been solved with longer IDs)

A practical demonstration

The PGP/GPG Encrypted Mail Protocol

- Pretty Good Privacy, a.k.a. PGP was designed back in 1995 by Phil Zimmermann as the first attempt to provide strong crypto to anyone
- Gnu Privacy Guard, a.k.a. GPG was developed by Werner Koch as a free alternative to PGP, and supports the same features.
- Possible to encrypt and sign any file, the most typical use is to encrypt and sign e-mail messages
- GPG is CLI only, but GUIs are available for all the most common mail clients: Thunderbird→Enigmail, Apple Mail→GPGSuite, Outlook→GPGOl

The PGP/GPG Encrypted Mail Protocol

Message preparation

- PGP/GPG messages encrypted and signed with an hybrid symmetric+asymmetric scheme for efficiency
- The signature and encryption actions are disjoint and the user may select whether to perform both of them, or only one
- The procedure goes as follows:
  1. The message is compressed (zip and bz2 algorithm supported)
  2. A random symmetric session key is generated for each recipient
  3. A copy of the message is encrypted for each symmetric key
  4. Each symmetric key is encrypted with the recipient public key
  5. The messages are bundled together and the hash of the bundle is signed
- Current implementations support (default in red)
  - Asymm. Ciphers: RSA,**ElGamal/DSA**
  - Symm. Ciphers: AES,Camellia,3DES,**CAST5**,Twofish,Blowfish
  - Hashes: SHA-1,**SHA-2**,RIPEMD-160

The PGP/GPG Encrypted Mail Protocol

E-mail Message preparation

- PGP/GPG messages are expressed in the same packet-based format as the WoT certificates
- Once the signed message bundle is ready, its encoding is ready to be embedded in a common e-mail (all characters are 7-bit ASCII)
- The first way to embed the message bundle in an e-mail is to enclose between two text marker [a]: the PGP/GPG mail client will match it
- As this tends to clutter the message, especially in the signature-only case, it is possible to employ a `application/pgp-encrypted` MIME section followed by an `application/octet-stream` one for encrypted messages and an `application/pgp-signature` to delimit the signature

[a] looking like `-----BEGIN PGP SIGNED MESSAGE-----`

The PGP/GPG Encrypted Mail Protocol

Small practical manual

- You can generate a key interactively with `gpg --gen-key`
- Encrypt a file `gpg --recipient "John Doe" -e <file>`
  - Add `--armor` if you need the output to be printable (e.g. mail)
  - Add `-s` if you also want to sign the file
- Decrypt with `gpg -d <file>`, gpg will pick the correct private key automatically
- Export a key to a keyserver with `gpg --send-key --keyserver <server name> <keyID>`, import with `gpg --recv-key --keyserver <keyID>`
- List all the certificates in your keyring with `gpg --list-keys`, check the certificate validity with `gpg --check-sigs`