HowTo (2) Certificats i GnuPG

Curs 2015 - 2016

Documentació a consultar:

https://www.gnupg.org/documentation/
https://www.gnupg.org/gph/en/manual.pdf

Descripcions

Software Aplications

GnuPG

GNU Privacy Guard (GnuPG or GPG) is a free software replacement for Symantec's PGP cryptographic software suite. GnuPG is compliant with RFC 4880, which is the IETF standards track specification of OpenPGP. Modern versions of PGP and Veridis' Filecrypt are interoperable with GnuPG and other OpenPGP-compliant systems.

As of versions 2.0.26 and 1.4.18, GnuPG supports the following algorithms:

- Pubkey: RSA, ElGamal, DSA
- Cipher: IDEA (from 1.4.13/2.0.20), 3DES, CAST5, Blowfish, AES-128, AES-192, AES-256, Twofish, Camellia-128, Camellia-192, Camellia-256 (from 1.4.10/2.0.12)
- Hash: MD5, SHA-1, RIPEMD-160, SHA-256, SHA-384, SHA-512, SHA-224
- Compression: Uncompressed, ZIP, ZLIB, BZIP2

GnuPG 2.1 series is announced to support elliptic curve cryptography (ECDSA, ECDH and EdDSA). *[wiki]*

PGP

Pretty Good Privacy (PGP) is a data encryption and decryption computer program that provides cryptographic privacy and authentication for data communication. PGP is often used for signing, encrypting, and decrypting texts, e-mails, files, directories, and whole disk partitions and to increase the security of e-mail communications. It was created by Phil Zimmermann in 1991.*[wiki]*

Symetric Cryptography

DES

The Data Encryption Standard was once a predominant symmetric-key algorithm for the encryption of electronic data. It was highly influential in the advancement of modern cryptography in the academic world. Developed in the early 1970s at IBM and based on an earlier design by Horst Feistel.Was published as an official Federal Information Processing Standard (FIPS) for the United States in 1977.

DES is now considered to be insecure for many applications. This is mainly due to the 56-bit key size being too small; in January 1999, distributed.net and the Electronic Frontier Foundation collaborated to publicly break a DES key in 22 hours and 15 minutes.

The algorithm is believed to be practically secure in the form of Triple DES. *[wiki]*

3DES

In cryptography, Triple DES (3DES) is the common name for the **Triple Data Encryption Algorithm** (TDEA or Triple DEA) symmetric-key **block cipher**, which applies the Data Encryption Standard (DES) cipher algorithm three times to each data block.

The original DES cipher's key size of 56 bits was generally sufficient when that algorithm was designed, but the availability of increasing computational power made brute-force attacks feasible. Triple DES provides a relatively simple method of increasing the key size of DES to protect against such attacks, without the need to design a completely new block cipher algorithm.

In recent years, the cipher has been superseded by the Advanced Encryption Standard (AES). *[wiki]*

CAST5

By default, GnuPG uses the CAST5 symmetrical algorithm.
In cryptography, CAST-128 (alternatively CAST5) is a symmetric-key **block cipher** used in a number of products, notably as the default cipher in some versions of GPG and PGP. It has also been approved for Canadian government use by the Communications Security Establishment. The algorithm was created in 1996 by Carlisle Adams and Stafford Tavares using the CAST design procedure. *[wiki]*

AES

The Advanced Encryption Standard (AES), also known as Rijndael[(its original name), is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

AES is based on the Rijndael cipher[5] developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, who submitted a proposal to NIST during the AES selection process. Rijndael is a family of ciphers with different key and block sizes.

For AES, NIST selected three members of the Rijndael family, each with a block size of 128 bits, but three different key lengths: 128, 192 and 256 bits.

AES has been adopted by the U.S. government and is now used worldwide. It supersedes the Data Encryption Standard (DES), which was published in 1977. The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.ES became effective as a federal government standard on May 26, 2002. *[wiki]*

IDEA

In cryptography, the **International Data Encryption Algorithm** (IDEA), originally called Improved Proposed Encryption Standard (IPES), is a symmetric-key **block cipher** designed by James Massey of ETH Zurich and Xuejia Lai and was first described in 1991. The algorithm was intended as a replacement for the Data Encryption Standard (DES). IDEA is a minor revision of an earlier cipher, Proposed Encryption Standard (PES). *[wiki]*

Asymetric Cryptography

DSA

The Digital Signature Algorithm Standard. PublicKey **signature algorithm**.
The Digital Signature Algorithm (DSA) is a Federal Information Processing Standard for digital signatures. It was proposed by the National Institute of Standards and Technology (NIST) in August 1991 for use in their Digital Signature Standard (DSS) and adopted as FIPS 186 in 1993. Four revisions to the initial specification have been released: FIPS 186-1 in 1996, FIPS 186-2 in 2000, FIPS 186-3 in 2009, and FIPS 186-4 in 2013. *[wiki]*

RSA

The **Rivest-Shamir-Adleman** cryptosystem, a cryptosystem for public-key encryption. RSA is one of the first practical public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, the encryption key is public and differs from the decryption key which is kept secret.

In RSA, this asymmetry is based on the practical difficulty of factoring the product of two large prime numbers, the factoring problem. RSA is made of the initial letters of the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman, who first publicly described the algorithm in 1977. Clifford Cocks, an English mathematician working for the UK intelligence agency GCHQ, had developed an equivalent system in 1973, but it was not declassified until 1997.

A user of RSA creates and then publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime numbers can feasibly decode the message.

RSA is a relatively slow algorithm, and because of this it is less commonly used to directly encrypt user data. More often, RSA passes encrypted shared keys for symmetric key cryptography which in turn can perform bulk encryption-decryption operations at much higher speed. *[wiki]*

ElGamal

In cryptography, the ElGamal encryption system is an **asymmetric key encryption algorithm** for public-key cryptography which is based on the Diffie–Hellman key exchange. It was described by Taher Elgamal in 1985. ElGamal encryption is used in the free GNU Privacy Guard software, recent versions of PGP, and other cryptosystems. The DSA (Digital Signature Algorithm) is a variant of the ElGamal signature scheme, which should not be confused with ElGamal encryption.

ElGamal encryption can be defined over any cyclic group G. Its security depends upon the difficulty of a certain problem in G related to computing discrete logarithms. ElGamal encryption consists of three components: the key generator, the encryption algorithm, and the decryption algorithm. *[wiki]*

Diffie-Hellman

Diffie–Hellman **key agreement** itself is a non-authenticated key-agreement protocol, it provides the basis for a variety of authenticated protocols, and is used to provide forward secrecy in Transport Layer Security's ephemeral modes.

Diffie–Hellman **key exchange** (D–H) is a specific method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols as originally conceptualized by Ralph Merkle. D–H is one of the earliest practical examples of public key exchange implemented within the field of cryptography. Traditionally, secure encrypted communication between two parties required that they first exchange keys by some secure physical channel, such as paper key lists transported by a trusted courier.

The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

Diffie–Hellman is used to secure a variety of Internet services. However, research published in October 2015 suggests that the parameters in use for many D-H Internet applications at that time are not strong enough to prevent compromise by very well-funded attackers, such as the security services of large governments.[3]

The scheme was first published by Whitfield Diffie and Martin Hellman in 1976.[2] By 1975, James H. Ellis,[4] Clifford Cocks and Malcolm J. Williamson within GCHQ, the British signals intelligence agency, had previously shown how public-key cryptography could be achieved; however, their work was kept secret until 1997. *[wiki]*

Hash Functions

MD5

The MD5 **message-digest algorithm** is a widely used cryptographic hash function producing a 128-bit (16-byte) hash value, typically expressed in text format as a 32 digit hexadecimal number. MD5 has been utilized in a wide variety of cryptographic applications, and is also commonly used to verify data integrity.

MD5 was designed by Ronald Rivest in 1991 to replace an earlier hash function, MD4. The source code in RFC 1321 contains a "by attribution" RSA license.

In 1996 a flaw was found in the design of MD5. While it was not deemed a fatal weakness at the time, cryptographers began recommending the use of other algorithms, such as SHA-1—which has since been found to be vulnerable as well.

In 2004 it was shown that MD5 is not collision resistant. As such, MD5 is not suitable for applications like SSL certificates or digital signatures that rely on this property for digital security. Also in 2004 more serious flaws were discovered in MD5, making further use of the algorithm for security purposes questionable; specifically, a group of researchers described how to create a pair of files that share the same MD5 checksum.

U.S. government applications now require the SHA-2 family of hash functions.[12] In 2012, the Flame malware exploited the weaknesses in MD5 to fake a Microsoft digital signature. *[wiki]*

SHA

In cryptography, SHA-1 (**Secure Hash Algorithm** 1) is a cryptographic hash function designed by the United States National Security Agency and is a U.S. Federal Information Processing Standard published by the United States NIST SHA-1 produces a 160-bit (20-byte) hash value known as a message digest. A SHA-1 hash value is typically rendered as a hexadecimal number, 40 digits long.

SHA-1 is no longer considered secure against well-funded opponents. In 2005, cryptanalysts found attacks on SHA-1 suggesting that the algorithm might not be secure enough for ongoing use, and since 2010 many organizations have recommended its replacement by **SHA-2** or SHA-3. Microsoft, Google and Mozilla have all announced that their respective browsers will stop accepting SHA-1 SSL certificates by 2017.. [*wiki*]

Public Key model

PKI

A **public key infrastructure** (PKI) is a set of hardware, software, people, policies, and procedures needed to create, manage, distribute, use, store, and revoke digital certificates[1] and manage public-key encryption. The purpose of a PKI is to facilitate the secure electronic transfer of information for a range of network activities such as e-commerce, internet banking and confidential email. It is required for activities where simple passwords are an inadequate authentication method and more rigorous proof is required to confirm the identity of the parties involved in the communication and to validate the information being transferred.

In cryptography, a PKI is an arrangement that binds public keys with respective user identities by means of a certificate authority (CA). The user identity must be unique within each CA domain. The third-party validation authority (VA) can provide this information on behalf of the CA. The binding is established through the registration and issuance process. Depending on the assurance level of the binding, this may be carried out by software at a CA or under human supervision. [*wiki*]

Web of trust

In cryptography, a **web of trust** is a concept used in PGP, GnuPG, and other OpenPGP-compatible systems to establish the authenticity of the binding between a public key and its owner. Its decentralized trust model is an alternative to the centralized trust model of a public key infrastructure (PKI), which relies exclusively on a certificate authority (or a hierarchy of such). As with computer networks, there are many independent webs of trust, and any user (through their identity certificate) can be a part of, and a link between, multiple webs. [*wiki*]

File format

PEM

The .pem filename extension is used for a Base64-encoded X.509 certificate. [*wiki*]

Certificate filename extensions

Common filename extensions for X.509 certificates are:

- .pem – (Privacy-enhanced Electronic Mail) Base64 encoded DER certificate, enclosed between "-----BEGIN CERTIFICATE-----" and "-----END CERTIFICATE-----"

- .cer, .crt, .der – usually in binary DER form, but Base64-encoded certificates are common too (see .pem above)

- .p7b, .p7c – PKCS#7 SignedData structure without data, just certificate(s) or CRL(s)
- .p12 – PKCS#12, may contain certificate(s) (public) and private keys (password protected)
- .pfx – PFX, predecessor of PKCS#12 (usually contains data in PKCS#12 format, e.g., with PFX files generated in IIS)

PKCS#7 is a standard for signing or encrypting (officially called "enveloping") data. Since the certificate is needed to verify signed data, it is possible to include them in the SignedData structure. A .P7C file is a degenerated SignedData structure, without any data to sign.

PKCS#12 evolved from the *personal information exchange* (PFX) standard and is used to exchange public and private objects in a single file.

X509

In cryptography, X.509 is an ITU-T standard for a public key infrastructure (PKI) and Privilege Management Infrastructure (PMI). X.509 specifies, amongst other things, standard formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm. [*wiki*]

Structure of a certificate

The structure foreseen by the standards is expressed in a formal language, namely Abstract Syntax Notation One.
The structure of an X.509 v3 digital certificate is as follows:

- Certificate
  - Version Number
  - Serial Number
  - Signature Algorithm ID
  - Issuer Name
  - Validity period
    - Not Before
    - Not After
  - Subject name
  - Subject Public Key Info
    - Public Key Algorithm
    - Subject Public Key
  - Issuer Unique Identifier (optional)
  - Subject Unique Identifier (optional)
  - Extensions (optional)
    - ...

- Certificate Signature Algorithm
- Certificate Signature

Other

Alice & Bob

Alice & Bob [*wiki*]

Jane & Dave

Jane & Dave [wiki]

DSA Signing

[wiki]

Key generation

Key generation has two phases. The first phase is a choice of *algorithm parameters* which may be shared between different users of the system, while the second phase computes public and private keys for a single user.

Parameter generation

- Choose an approved cryptographic hash function H. In the original DSS, H was always SHA-1, but the stronger SHA-2 hash functions are approved for use in the current DSS.[5][9] The hash output may be truncated to the size of a key pair.
- Decide on a key length L and N. This is the primary measure of the cryptographic strength of the key. The original DSS constrained L to be a multiple of 64 between 512 and 1024 (inclusive). NIST 800-57 recommends lengths of 2048 (or 3072) for keys with security lifetimes extending beyond 2010 (or 2030), using correspondingly longer N.[10] FIPS 186-3 specifies L and N length pairs of (1024,160), (2048,224), (2048,256), and (3072,256).[4] N must be less than or equal to the output length of the hash H.
- Choose an N-bit prime q.
- Choose an L-bit prime modulus p such that $p − 1$ is a multiple of q.
- Choose g, a number whose multiplicative order modulo p is q. This may be done by setting $g = h^{(p-1)/q} \bmod p$ for some arbitrary h ($1 < h < p − 1$), and trying again with a different h if the result comes out as 1. Most choices of h will lead to a usable g; commonly $h = 2$ is used.

The algorithm parameters (p, q, g) may be shared between different users of the system.

Per-user keys

Given a set of parameters, the second phase computes private and public keys for a single user:

- Choose a secret key x by some random method, where $0 < x < q$.
- Calculate the public key $y = g^x \bmod p$.

There exist efficient algorithms for computing the modular exponentiations $h^{(p-1)/q} \bmod p$ and $g^x \bmod p$, such as exponentiation by squaring.

Signing

Let H be the hashing function and m the message:

- Generate a random per-message value k where $0 < k < q$
- Calculate $r = \left(g^k \bmod p\right) \bmod q$
- In the unlikely case that $r = 0$, start again with a different random k
- Calculate $s = k^{-1} \left(H\left(m\right) + xr\right) \bmod q$
- In the unlikely case that $s = 0$, start again with a different random k
- The signature is $\left(r, s\right)$

The first two steps amount to creating a new per-message key. The modular exponentiation here is the most computationally expensive part of the signing operation, and it may be computed before the message hash is known. The modular inverse $k^{-1} \bmod q$ is the second most expensive part, and it may also be computed before the message hash is known. It may be computed using the extended Euclidean algorithm or using Fermat's little theorem as $k^{q-2} \bmod q$.

Verifying

- Reject the signature if $0 < r < q$ or $0 < s < q$ is not satisfied.
- Calculate $w = s^{-1} \bmod q$
- Calculate $u_1 = H\left(m\right) \cdot w \bmod q$
- Calculate $u_2 = r \cdot w \bmod q$
- Calculate $v = \left(g^{u_1} y^{u_2} \bmod p\right) \bmod q$
- The signature is invalid unless $v = r$

DSA is similar to the ElGamal signature scheme.

Correctness of the algorithm

The signature scheme is correct in the sense that the verifier will always accept genuine signatures. This can be shown as follows:
First, if $g = h^{(p-1)/q} \bmod p$ it follows that $g^q \equiv h^{p-1} \equiv 1 \pmod{p}$ by Fermat's little theorem.
Since $g > 1$ and q is prime, g must have order q.
The signer computes

$$s = k^{-1}(H(m) + xr) \bmod q$$

Thus

$$\begin{aligned} k &\equiv H(m)s^{-1} + xrs^{-1} \\ &\equiv H(m)w + xrw \pmod{q} \end{aligned}$$

Since g has order q (mod p) we have

$$\begin{aligned} g^k &\equiv g^{H(m)w} g^{xrw} \\ &\equiv g^{H(m)w} y^{rw} \\ &\equiv g^{u1} y^{u2} \pmod{p} \end{aligned}$$

Finally, the correctness of DSA follows from

$$\begin{aligned} r &= (g^k \bmod p) \bmod q \\ &= (g^{u1} y^{u2} \bmod p) \bmod q \\ &= v \end{aligned}$$

RSA System

[wiki]

Operation

The RSA algorithm involves four steps: key generation, key distribution, encryption and decryption.
RSA involves a *public key* and a *private key*. The public key can be known by everyone and is used for encrypting messages. The intention is that messages encrypted with the public key can only be decrypted in a reasonable amount of time using the private key.
The basic principle behind RSA is the observation that it is practical to find three very large positive integers *e*,*d* and *n* such that with modular exponentiation for all *m*:

$$(m^e)^d \mod n = m$$

and that even knowing *e* and *n* or even *m* it can be extremely difficult to find *d*.
Additionally, for some operations it is convenient that the order of the two exponentiations can be changed and that this relation also implies:

$$(m^d)^e \mod n = m$$

Key distribution

To enable Bob to send his encrypted messages, Alice transmits her public key (*n*, *e*) to Bob via a reliable, but not necessarily secret route, and keeps the private key *d* secret and this is never revealed to anyone. Once distributed the keys can be reused over and over.

Encryption

Bob then wishes to send message *M* to Alice.
He first turns *M* into an integer *m*, such that $0 \leq m < n$ and gcd(*m*, *n*) = 1 by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext *c*, using the public key *e* of Alice, corresponding to

$$c \equiv m^e \mod n$$

This can be done efficiently, even for 500-bit numbers, using modular exponentiation. Bob then transmits *c* to Alice.

Decryption

Alice can recover *m* from *c* by using her private key exponent *d* by computing

$$c^d \equiv (m^e)^d \equiv m \mod n$$

Given *m*, she can recover the original message *M* by reversing the padding scheme.

Key generation

The keys for the RSA algorithm are generated the following way:

1. Choose two distinct prime numbers *p* and *q*.
   - For security purposes, the integers *p* and *q* should be chosen at random, and should be similar in magnitude but 'differ in length by a few digits'[2] to make factoring harder. Prime integers can be efficiently found using a primality test.
2. Compute *n* = *pq*.
   - *n* is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.
3. Compute φ(*n*) = φ(*p*)φ(*q*) = (*p* − 1)(*q* − 1) = *n* − (*p* + *q* − 1), where φ is Euler's totient function. This value is kept private.
4. Choose an integer *d* such that 1 < *d* < φ(*n*) and gcd(*d*, φ(*n*)) = 1; i.e., *d* and φ(*n*) are coprime.
5. Determine *e* as $e \equiv d^{-1}$ (mod φ(*n*)); i.e., *e* is the modular multiplicative inverse of *d* (modulo φ(*n*))

- This is more clearly stated as: solve for *e* given *d*⋅*e* ≡ 1 (mod φ(*n*))
- *e* having a short bit-length and small Hamming weight results in more efficient encryption – most commonly $2^{16} + 1$ = 65,537. However, much smaller values of *e* (such as 3) have been shown to be less secure in some settings.[13]
- *e* is released as the public key exponent.
- *d* is kept as the private key exponent.

The *public key* consists of the modulus *n* and the public (or encryption) exponent *e*. The *private key* consists of the modulus *n* and the private (or decryption) exponent *d*, which must be kept secret. *p*, *q*, and φ(*n*) must also be kept secret because they can be used to calculate *d*.

- An alternative, used by PKCS#1, is to choose *d* matching *de* ≡ 1 (mod λ) with λ = lcm(*p* − 1, *q* − 1), where lcm is the least common multiple. Using λ instead of φ(*n*) allows more choices for *d*. λ can also be defined using the Carmichael function, λ(*n*).

Since any common factors of (p-1) and (q-1) are present in the factorisation of p*q-1,[14] it is recommended that (p-1) and (q-1) have only very small common factors, if any besides the necessary 2.[15][2][16]

Example

Here is an example of RSA encryption and decryption. The parameters used here are artificially small, but one can also use OpenSSL to generate and examine a real keypair.

1. Choose two distinct prime numbers, such as
2. $p = 61$ and $q = 53$
3. Compute *n* = *pq* giving
4. $n = 61 \times 53 = 3233$
5. Compute the totient of the product as φ(*n*) = (*p* − 1)(*q* − 1) giving
6. $\varphi(3233) = (61 - 1)(53 - 1) = 3120$
7. Choose any number 1 < *d* < 3120 that is coprime to 3120. Choosing a prime number for *d* leaves us only to check that *d* is not a divisor of 3120.
8. Let $d = 2753$
9. Compute *e*, the modular multiplicative inverse of *d* (mod φ(*n*)) yielding,
10. $e = 17$

11. Worked example for the modular multiplicative inverse:
12. $d \times e \mod \varphi(n) = 1$
13. $2753 \times 17 \mod 3120 = 1$

The **public key** is (n = 3233, e = 17). For a padded plaintext message m, the encryption function is

$$c(m) = m^{17} \mod 3233$$

The **private key** is (d = 2753). For an encrypted ciphertext c, the decryption function is

$$m(c) = c^{2753} \mod 3233$$

For instance, in order to encrypt m = 65, we calculate

$$c = 65^{17} \mod 3233 = 2790$$

To decrypt c = 2790, we calculate

$$m = 2790^{2753} \mod 3233 = 65$$

Both of these calculations can be computed efficiently using the square-and-multiply algorithm for modular exponentiation. In real-life situations the primes selected would be much larger; in our example it would be trivial to factor n, 3233 (obtained from the freely available public key) back to the primes p and q. Given e, also from the public key, we could then compute d and so acquire the private key.

Practical implementations use the Chinese remainder theorem to speed up the calculation using modulus of factors (mod pq using mod p and mod q).

The values d_p, d_q and q_{inv}, which are part of the private key are computed as follows:

$$d_p = d \mod (p-1) = 2753 \mod (61-1) = 53$$
$$d_q = d \mod (q-1) = 2753 \mod (53-1) = 49$$
$$q_{\text{inv}} = q^{-1} \mod p = 53^{-1} \mod 61 = 38$$
$$\Rightarrow (q_{\text{inv}} \times q) \mod p = 38 \times 53 \mod 61 = 1$$

Here is how d_p, d_q and q_{inv} are used for efficient decryption. (Encryption is efficient by choice of a suitable d and e pair)

$$m_1 = c^{d_p} \mod p = 2790^{53} \mod 61 = 4$$
$$m_2 = c^{d_q} \mod q = 2790^{49} \mod 53 = 12$$
$$h = (q_{\text{inv}} \times (m_1 - m_2)) \mod p = (38 \times -8) \mod 61 = 1$$
$$m = m_2 + h \times q = 12 + 1 \times 53 = 65$$

Signing messages

Suppose Alice uses Bob's public key to send him an encrypted message. In the message, she can claim to be Alice but Bob has no way of verifying that the message was actually from Alice since anyone can use Bob's public key to send him encrypted messages. In order to verify the origin of a message, RSA can also be used to sign a message.

Suppose Alice wishes to send a signed message to Bob. She can use her own private key to do so. She produces a hash value of the message, raises it to the power of d (modulo n) (as she does when decrypting a message), and attaches it as a "signature" to the message. When Bob receives the signed message, he uses the same hash algorithm in conjunction with Alice's public key. He raises the signature to the power of *e* (modulo *n*) (as he does when encrypting a message), and compares the resulting hash value with the message's actual hash value. If the two agree, he knows that the author of the message was in possession of Alice's private key, and that the message has not been tampered with since.

Diffie - Hellman

[wiki]

Description

General overview

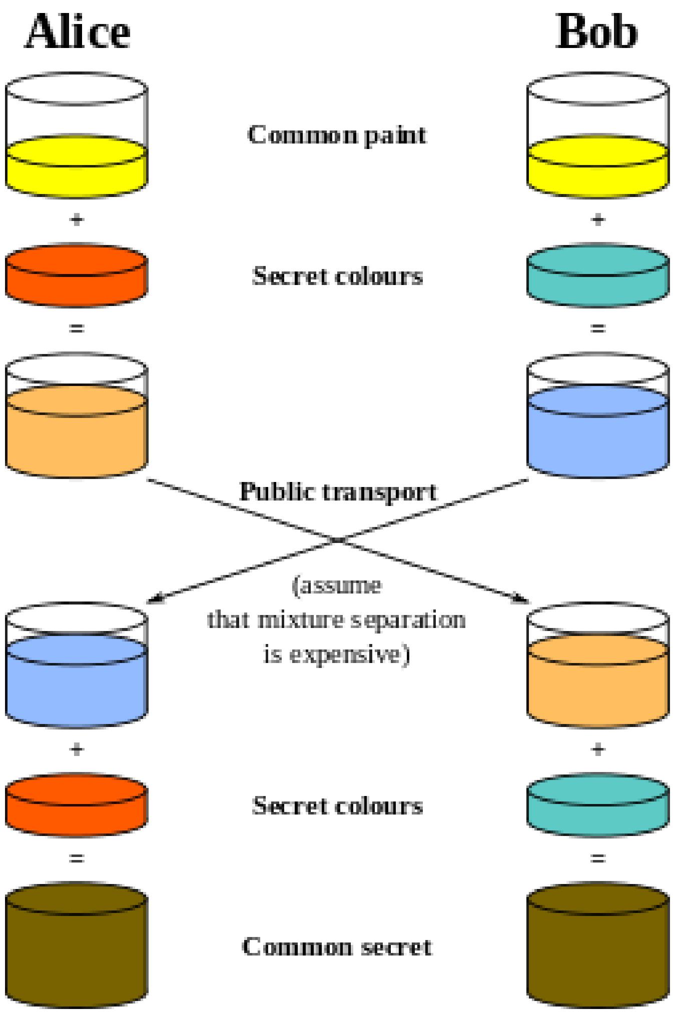

Illustration of the Diffie–Hellman Key Exchange

Diffie–Hellman Key Exchange establishes a shared secret between two parties that can be used for secret communication for exchanging data over a public network. The following conceptual diagram illustrates the general idea of the key exchange by using colors instead of very large numbers.
The process begins by having the two parties, Alice and Bob, agree on an arbitrary starting color that does not need to be kept secret (but should be different every time[8]); in this

example the color is yellow. Each of them selects a secret color–red and aqua respectively–that they keep to themselves. The crucial part of the process is that Alice and Bob now mix their secret color together with their mutually shared color, resulting in orange and blue mixtures respectively, then publicly exchange the two mixed colors. Finally, each of the two mix together the color they received from the partner with their own private color. The result is a final color mixture (brown) that is identical to the partner's color mixture.
If another party (usually named *Eve* in cryptology publications, Eve being a third-party who is considered to be an eavesdropper) had been listening in on the exchange, it would be computationally difficult for that person to determine the common secret color; in fact, when using large numbers rather than colors, this action is impossible for modern supercomputers to do in a reasonable amount of time.

Cryptographic explanation

The simplest and the original implementation of the protocol uses the multiplicative group of integers modulo *p*, where *p* is prime, and *g* is a primitive root modulo *p*. These two values are chosen in this way to ensure that the resulting shared secret can take on any value from 1 to *p*–1. Here is an example of the protocol, with non-secret values in blue, and secret values in **red**.

1. Alice and Bob agree to use a modulus $p = 23$ and base $g = 5$ (which is a primitive root modulo 23).
2. Alice chooses a secret integer $\mathbf{a} = \mathbf{6}$, then sends Bob $A = g^{a}$ mod p
   - $A = 5^{6}$ mod $23 = 8$
3. Bob chooses a secret integer $\mathbf{b} = \mathbf{15}$, then sends Alice $B = g^{b}$ mod p
   - $B = 5^{15}$ mod $23 = 19$
4. Alice computes $\mathbf{s} = B^{a}$ mod p
   - $\mathbf{s} = 19^{6}$ mod $23 = \mathbf{2}$
5. Bob computes $\mathbf{s} = A^{b}$ mod p
   - $\mathbf{s} = 8^{15}$ mod $23 = \mathbf{2}$
6. Alice and Bob now share a secret (the number **2**).

Both Alice and Bob have arrived at the same value s, because, under mod p,

$$A^b = g^{ab} = g^{ba} = B^a \ (mod\ p)$$ [9]

Note that only *a*, *b*, and *(g^{ab} mod p = g^{ba} mod p)* are kept secret. All the other values – *p*, *g*, g^a mod *p*, and g^b mod *p* – are sent in the clear. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.
Of course, much larger values of *a*, *b*, and *p* would be needed to make this example secure, since there are only 23 possible results of *n* mod 23. However, if *p* is a prime of at least 600 digits, then even the fastest modern computers cannot find *a* given only *g*, *p* and g^a mod *p*. Such a problem is called the discrete logarithm problem.[3] The computation of g^a mod *p* is known as modular exponentiation and can be done efficiently even for large numbers. Note that *g* need not be large at all, and in practice is usually a small integer (like 2, 3, ...).

Generalization to finite cyclic groups

Here is a more general description of the protocol:[10]

1. Alice and Bob agree on a finite cyclic group G of order n and a generating element g in G. (This is usually done long before the rest of the protocol; g is assumed to be known by all attackers.) The group G is written multiplicatively.
2. Alice picks a random natural number a, where $1 \le a < n$, and sends g^a to Bob.
3. Bob picks a random natural number b, which is also $1 \le b < n$, and sends g^b to Alice.
4. Alice computes $(g^b)^a$.
5. Bob computes $(g^a)^b$.

Both Alice and Bob are now in possession of the group element g^{ab}, which can serve as the shared secret key. The group G satisfies the requisite condition for secure communication if there is not an efficient algorithm for determining whether $g^{ab} = g^c$ given g^a, g^b, and g^c for some $c \in G$.

Secrecy chart

The chart below depicts who knows what, again with non-secret values in blue, and secret values in **red**. Here Eve is an eavesdropper—she watches what is sent between Alice and Bob, but she does not alter the contents of their communications.

- g = public (prime) base, known to Alice, Bob, and Eve. $g = 5$
- p = public (prime) modulus, known to Alice, Bob, and Eve. $p = 23$
- $\boldsymbol{a}$ = Alice's private key, known only to Alice. $\boldsymbol{a} = \mathbf{6}$
- $\boldsymbol{b}$ = Bob's private key known only to Bob. $\boldsymbol{b} = \mathbf{15}$
- A = Alice's public key, known to Alice, Bob, and Eve. $A = g^{\boldsymbol{a}} \bmod p = 8$
- B = Bob's public key, known to Alice, Bob, and Eve. $B = g^{\boldsymbol{b}} \bmod p = 19$

Descripció general

Certificats

Mecanisme de certificació:

1. PKI Public key infrastructure: Basat en signatures de una autoritat certificadora CA.
2. Confiança en les claus dels altres: web trust scheme. Són self-signed certificats.

Perquè usar certificats i no simplement usar les claus públiques dels altres per comunicar-nos amb ells?

Si rebem la clau pública de alice i en realitat no és la de alice sinó la d'un altre, aquest pot desxifrar amb la seva clau privada el que li diem a l'alice. La qüestió és, "*com se que la clau pública de fulanito és realment de fulanito?*" O bé em refio del que em diu o bé ho contrasto amb una entitat certificadora.

Public key certificate

Document amb firma digital que conté: (public key + identity) d'algú. El certificat està signat per alguna autoritat que emet certificats i garanteix que fulanito és tal i que la seva clau pública és tal.

Un certificat és associar la clau pública d'algú amb la seva identitat. El certificat es firma per una CA. Per firmar es fa un hash (de la clau pública de fulanito) i se li aplica la clau privada de la CA (generalment algorisme RSA).

El certificat actual més usat és el **X.509**. Permet el seu ús en LDAP (X.500 schema). Software per generar certificats: OpenSSL,OpenCA, EJBCA Openvpn, Openssh

Public Key Infraestructure: PKI

En una estructura PKI una tercera part TTP (**trusted tried party**) signa el certificat per donar-li validesa. Es tracta d'una entitat certificadora **CA**.

En una estructura web of trust el certificat l'envia un paio ite'l creus o no. Són self-signed certificats. Un exemples és el PGP, OpenPGP il GNUPG.

En una estructura PKI el certificat està signat per una autoritat certificadora. Qui certifica aquesta? una altra entitat de nivell superior. I així fins al nivell arrel. No existeix una única arrel. Els navegadors porten implementats diverses entitats arrel.

Es confiarà en un certificat si en algun punt de la cadena (cap al'arrel) es troba una entitat en la
que el receptor SI hi confia.

NO es xequegen les CA per preguntar-li si el certificat és vàlid o no (massa temps d'espera) sinó que els certificats poden incloure la cadena de certificacions fins la seva arrel i es d'esperar que el receptor trobarà en algun punt d'aquesta cadena una entitat en la que confia.

Per tant, un certificat pot incloure a més a més un conjunt d'altres certificats (el de les entitats
que l'han certificat des de l'arrel). **Certificate hierarchy.**

Com es certifica un certificat?

- La CA   firma la clau pública del fulanito amb la seva clau privada (es pot comprovar la validesa del certificat usant la clau pública del CA).
- La CA pot incloure la seva clau pública signada per una entitat superior, això permet validar el certificat Si cal inclou també la clau pública de l'entitat superior-superior que ha signat la superior... fins arribar a una entitat en la que es confia o en la que el navegador ja esta preconfigurat.

*** nota: mirar exemple de certificat X.509 de la wiki o mirar un certificat real d'exemple

X.598

X.509 estandard de certificat de PKI Public Key Infrastructure.
Basat en una estructura jeràrquica de entitats certidficadores CA.
Clau assimètrica. Clau publica / clau privada. Correspon a la clau pública signnada per una CA i inclou indormació de la identitat del propietari del certificat.

Tipus de certificats

.cer	CER encoded certificate. seqüències de certificat
.der	DER encoded certificate
.PEM	Privacy Ennhanced Mail. Base64 encoded DER.
.P7b	
.P7c	PKCS#7 certificat
.PFX	
.P12	PKCS#12 pot contenir certificats publics i claus privades

Criptografia asimatrica:

----
algorisme de clau publica/privada. encriptar i signar.
RSA Rivest, Shamir, Adleman (del MIT)
No van pensar a patentar-lo fora els usa.Ara ja patent caducada.
M - Epublick --> M1 --> Dprivadak - M

Criptografia Simetrica:

E, D: Algosrismes coneguts d'encriptar i desemcriptar
K: Clau idem emisor i receptor
M - Ek(M) --> M1 --> Dk(M1) - M

Algorismes de clau simètrica més fàcils i ràpids de calcular. Claus més petites.
Per això s'utilitza en les comunicacions en 'cru'. IPsec, ssh, https, (totes les s finals), TLS/SSL, etc.

DES: problema de fiabilitat de clau i dubtes de porta trasera.
DES3: més utilitzat. Clau reforçada..
AES: concurs public (uns belgues). Per ser el nou estàndard. 128bytes 196bytes, 256bytes

Hash del missatge:

MD5:
SHA1, SHA2

GnuPG

Funcions bàsiques

Crear i llistar claus

--key-gen --gen-revoke --list-keys --list-public-keys --list-private-keys

1. Crear un parell de claus. usuari *gpere*. i usuari *ganna*. Identificar amb nom, descripció i email.
2. Crear certificat de revocació usuari *gpere*.
3. Llistar les claus identificant-les per el ID, de clau o per qualsevol dels camps introduïts (nom, descripció o email).
    - ❏ Llistar la(s) public keys.
    - ❏ Llistar la(s) private keys.

```
pub   2048R/EE94E064 2016-02-11 [expires: 2026-02-08]
uid                  gpere pou prat (el famos pere perico) <gpere@localhost.localdomain>
sub   2048R/1AF02D87 2016-02-11 [expires: 2026-02-08]
```

Exportar public key

--export --output --armor

4. Exportar una clau pública (per que la pugui importar un altre suauri). exportar *ganna*.
    - ❏ Exportar en un fitxer en format binari comprimit.
    - ❏ Exportar en un fitxer ascii (PEM?).
5. Exportar de l'usuari *gpere* les seves claus i desar-les. No caldrà importar-les en lloc de moment.

Importar i signar public key

--import --list-keys --edit-key fpr sign check

6. Importar una clau pública d'un altre usuari al propi anell de claus. *gpere* importa la clau de *ganna*.
    - ❏ Llistar les claus de l'anell de claus de *gpere*.
    - ❏ Llistar els fingerprint de totes les claus.
7. Cal que l'usuari 'avali' la clau importada com a bona, per fer-ho cal signar-la. *gpere* signa com a vàlida la public key importada de *ganna*.
    - ❏ Editar la clau importada (*gpere* edita la public key de *ganna* que té al seu anell de claus).
    - ❏ Llistar el fingerprint de la clau importada. Verificar (per exemple trucant per telèfon a *ganna*) que el fingerrint és correcte.

❏ Signar com a valida (avalada) la clau importada. *gpere* signa la clau importada de *ganna* un cop ha verificat el fingerprint i confia en la validesa de la clau.
❏ Fer un check per comprovar les signatures de les claus.

Xifrar / Desxifrar

--encrypt --decrypt --recipient --output

8. Xifrar un contingut generant un nou document xifrat. Es xifra utilitzant la clau publica del destinatari. *gpere* xifra un fitxer (per exemple fstab) per enviar-lo a *ganna* xifrat.
   ❏ El nou document pot sortir per stdout o pot anar a un fitxer output (el més recomanable).
   ❏ El fitxer generat pot ser un bnari comprimit xifrat o un fitxer de text xifrat.
   ❏ Per a cada destinatari del missatge cal indicar l'opció "*--recipient*" que identifica al destinatari i per tant indica la clau pública a usar per xifrar. Si s'envia a *ganna* usara la clau publica de *ganna* (que ja està a l'anell de claus de *gpere* perquè s'ha importat).
   ❏ Si el missatge es destina a un mateix (de *gpere* a *gpere*) cal també indicar el destinatari *gpere* amb l'opció "*--recipient*".
   ❏ Un mateix missatge es pot enviar a múltiples destinataris indicant múltiples "*--recipients*".
9. Xifrar el contingut d'un fitxer de text (per exemple passwd) generant un fitxer ascii xifrat. *gpere* el xifra destinant-lo a *gepre* i *ganna*.

10. Desxifrar un missatge. *ganna* desxifra el fitxer que ha generat *gpere* per a ella (el fstab i el passwd).
11. Desxifrar un missatge xifrat destinat a un mateix. *gpere* desxifra el del passwd. Observar que no pot desxifrar el del fstab perquè ell no n'era un "recipient".

Xifrat Simètric

--symetric

12. Usar una clau simètrica per xifrar un missatge. gpere envia un contingut xifrat simètricament a ganna.
13. Desxifrar un missatge xifrat simètricament. Cal disposar de la passfrase usada per xifrar. ganna desxifra el missatge rebut (un cop pere li a dit, per exemple per telèfon, la clau simetrica).

Signar / Verificar

--sign --clearsign --verify --decrypt --detach-sign

14. Signar documents. *gpere* envia a *ganna* el fitxer fstab signat. Pere usa la seva clau provada per signar els documents.
   ❏ gpere signa un document generant un document binari signat i comprimit (per exemple fstab)..

   - ❏ Genrerar un document signat en tetx pla, “cleartext”. Un document signat no té perquè ser xifrat ni comprimit. S’incorpora al final del document la signatura. *gpere* envia a *ganna* per exemple passwd.
15. Verificar un document signat. Nomès verificar si és vàlid o no, sense exttreure el document.
   - ❏ *ganna* verifica el document rebut de *gpere*, el fstab.
   - ❏ *ganna* verifica el document cleartetx rebut de *gpere*, el passwd.
16. Verificar i extreure el document. El missatge rebut pel destinatari conté conjuntament el document + la firma digital. Per extreure el document a part cal “desxifrar” (en realitat es verificar + extreure).
   - ❏ *ganna* verifica i extreu el contingut del fitxer fstab rebut.
   - ❏ *ganna* verifica i extreu el contingut del fitxer passwd rebut.

17. Enviar un missatge signat generant una signatura apart “detached”. Caldrà enviar per una banda el contingut (el missatge) i per l’altra la signatura. És a dir, en signar es genera un fitxer de signatura a part.
   - ❏ Generar una signatura detached. Per exemple *gpere* signa fstab generant una signatura a part.
   - ❏ El destinatari, *ganna*, verifica la signatura usant el contingut (fstab) i el fitxer de signatura detached.

Key Management

Gestió de múltiples IDs i Keys

--edit-key list adduid addkey toggle key id delkey deluid

1. Afegir a un usuari diverses identitats i subkeys.
   - ❏ Afegir a *gpere* tres identitats més i una quarta esborrable.
   - ❏ Afegir a gpere una subkey de cada tipus possible:
     (3) DSA (sign only)   (4) RSA (sign only)   (5) Elgamal (encrypt only)
     (6) RSA (encrypt only)
   - ❏ llistar les keys publiques i privades.
   - ❏ llistar les signatures
2. Eliminar identitats i subkeys.
   - ❏ Seleccionar i desseleccionar identitats i keys.
   - ❏ Eliminar de gpere la ultima identitat i la última subkey creades.

Revocar keys i Ids

--edit-key revkey revsig

3. Revocar keys. Les sub keys es poden revocar de de l’edició de keys seleccionat la key i revocant-la. Això permet (millor que no pas amb el delete key) que la clau pública contingui les key revocades i els altres coneguin que estan revocades.
   - ❏ Del llistat de sub keys de l’usuari *gpere* revocar la última subkey.

    - ❏ Llistar les keys i observar la revocació.
4. Revocar IDs: no es poden revocar iDs però si que es pot revocar la self-signature del ID. Això l'invalida.
    - ❏ Revocar (invalidar la signatura) de l'últim ID de l'usuari *gpere*.
    - ❏ Llistar les keys i Ids.

```
gpg> list

pub  2048R/EE94E064  created: 2016-02-11  expires: 2026-02-08  usage: SC
                     trust: ultimate      validity: ultimate
sub  2048R/1AF02D87  created: 2016-02-11  expires: 2026-02-08  usage: E
sub  2048D/98993206  created: 2016-02-12  expires: 2018-02-11  usage: S
sub  2048R/B81E8A43  created: 2016-02-12  expires: 2019-02-11  usage: S
sub  2048g/4D012BA0  created: 2016-02-12  expires: 2020-02-11  usage: E
sub  2048R/7B554A3B  created: 2016-02-12  expires: 2021-02-10  usage: E
This key was revoked on 2016-02-12 by RSA key EE94E064 gpere pou prat (el famos pere perico)
<gpere@localhost.localdomain>
sub  3072D/AAA293C7  created: 2016-02-12  revoked: 2016-02-12  usage: S
[ultimate] (1)  gpere pou prat (el famos pere perico) <gpere@localhost.localdomain>
[ unknown] (2)  ppere (en pere de l'escola del treball) <ppere@edt.cat>
[ revoked] (3)* perico (de l'espanyol) <perico@rcde.es>
```

Modificar Expiration Time

--edit-key expiration

5. Modificar el expiration time de:
    - ❏ La master keu de gpere de manera que sigui per sempre
    - ❏ De la segona subkey, fent-la per 20 anys.
    - ❏ Llistar les keys.

```
pub  2048R/EE94E064  created: 2016-02-11  expires: never       usage: SC
                     trust: ultimate      validity: ultimate
sub* 2048R/1AF02D87  created: 2016-02-11  expires: 2036-02-07  usage: E
```

Exportació de claus

--export --import

6. L'exportació de claus inclou totes les identitats i subclaus de l'usuari. La importació de claus es fa fent un **merge** amb les claus preexistents.
    - ❏ Així, si per exemple *ganna* importa les claus originals de gpere, contenen únicament una identitat i una clau Master i una subkey.
    - ❏ Importar a *ganna* la clau de *gpere* exportada inicialment. Llistar les claus de què disposa *ganna* al seu keyring.
7. Diferència entre eliminar i revocar. A *gpere* s'han creat diverses claus i identitats. Se n'han afegit, eliminat i revocat. L'usuaria *ganna* ha importat les claus inicials de *gpere*, si posteriorment importa de nou les claus actuals de *gpere* farà una 'barrega', un **merge** de unes i altres. És a dir, no es sobreescriuen sinó que es fusionen. Per

tant les keys i IDs que estiguessin abans a *gpere* (i que *ganna* els tingues) encara que s'esborrin a gpere continuaràn estant a ganna. Però si en lloc d'esborrar es revoquen llavors sí que *ganna* tindrà la informacio acurada de que aquestes keys i IDs ja no son vàlids.

- ❏ Exportar els certificats de *gpere*.
- ❏ Importar-los a *ganna*.

Trust / Validació de claus

trust

El concepte trust d'una clau descriu el grau en que el que importa la clau confia en la clau importada per avalar altres claus.

Suposem que A importa una clau pública de B que conté a més a més altres claus com per exemple de C i D. Són claus que B havia importat.

Confia A amb la clau pública de C i D que ha importat en importar B (anaven incorporats en l'anell de claus public que B ha exportat)?
Això s'estableix editant A la la clau B i establint un nivell de trruts de (4) 0 (5).

validity

Indica si la clau importada valida o no la identitat del posseïdor de la clau. Si A importa la clau pública de B pot signar la clau de B, això implica que validity és full. És a dir, que A té tota la certesa que la clau importada és realment la de B.

Si per exemple A importa la clau pública de B (que inclou la de C i D). Ara C val "trust:unknown , validity:undefined". Llavors A signa la clau de B (validity de B: full) i també estableix la confiaça (trust: a (4) o (5)).
Ara la clau de C és "trust:unknown validity:full". És a dir, ara A accepta com a vàlida, contrastada la identitat de C.

Atenció

trust i validity són conceptes claraments diferents. validitu s'obté signant una clau i indica que hi ha certesa que la clau és de qui diu ser. La clau de B és de B.
trust indica si confiem en el criteri d'un altre per donar per bones les claus. Si A incorpora un certificat de B, que signa perquè sap que és de B, no vol dir que confii en el criteri que té B per signat (avalar) altres claus.

8. Crear usuari *gmarta* i generar les seves claus. Exportar-les. Signar un document (per exemple /etc/passwd).
9. L'usuari *ganna* importa la clau publica de *gmarta*. Llistar les claus. Verificar la signatura del document. Editar la clau de *gmarta* (sense modificar-la) i observar el valor de trust i validity.
10. Usuari *gpere* valida el missatge signat que ha generat *gmarta*. No el pot validar perquè no té la seva clau pública.

11. La usuària *ganna* signa la clau que té de *gmarta*.
12. La usuària *ganna* exporta les seves claus i l'usuari gpere les importa. Llistar les claus. L'usuari gpere edita la clau de *ganna* i *gmarta* (sense modificar-les) per observar trust i validity.
13. L'usuari *gpere* valida el missatge signat per marta (el fitxer passwd). Indica que la signatura és la apropiada (perquè ara ja en té la clau pública) però que no pot verificar la identitat real de l'usuari que l'ha signat (no trust). Es a dir, esta ben signat però no se sap si per qui diu ser.
14. Finalment *gpere* modifica el valor de trust de la clau pública de *ganna* (no de *gmarta*!). Llistar les claus de *ganna* i *gmarta*. Observar que ara la clau de *gmarta* és "trust: unknown validity:full". O sigui, *gpere* que confia en *ganna* (trust i validity) accepta com a validity:full la clau que té de *gmarta*.
15. Ara *gpere* pot verificar la signatura del missatge signat per *gmarta*.