PGP

Pretty Good Privacy or PGP was written mostly by Phil Zimmermann.

He used the best available crypto algorithms as building blocks to create a system for enciphering both files and email.

It provides confidentiality and/or authentication.

It is independent of OS and machine.

It has a small number of easy-to-use commands.

It is freely available on the Internet.

Authentication is provided by SHA signed by either RSA or DSS.

Confidentiality is provided by encryption using either IDEA, 3DES, BLOWFISH, AES or TWOFISH, with a one-time key generated by the sender. They all use CFB mode.

PGP also provides compression, radix-64 conversion (for e-mail) and segmentation and reassembly of long messages.

The signature is generated before compression because:

(a) It is better to sign an uncompressed message so that you can store only the uncompressed message and signature for later verification. If you signed a compressed document, you would either have to store the compressed document or else recompress it at verification time.

(b) The compression algorithm is not deterministic. Different versions of produce different compressed files. Signing after compression would require the use of just one version of compression.

The message is enciphered after compression because the compressed message has less redundancy, so its cryptanalysis is harder.

The random numbers for generating session keys come from the timing of the users keystrokes.

Users may have more than one set of RSA keys (to change keys or to communicate with different sets of correspondents, say). Each public key is identified by its low-order 64 bits in messages sent to the recipient.

Each user of PGP has two data structures to hold keys: one for his own public/private key pairs and one to store the public keys of other users. These data structures are called the user's private-key ring and public-key ring.

The private keys are encrypted via a passphrase. SHA produces a 160-bit hash of the passphrase and 128 of these 160 bits are used as the key for CAST-128. Private keys are indexed either by their id (= low-order 64 bits) or by a user id.

The public keys are stored in a similar data structure, but which has additional fields for a timestamp and trust information.

Suppose Alice gets a public key for Bob from a source which has been compromised by Chuck, so that the key Alice thinks is Bob's really comes from Chuck. Then Chuck could send a message to Alice signed "Bob" and Alice would accept it as coming from Bob. Furthermore, Chuck could read any encrypted message from Alice to Bob.

One way to solve this problem would use X.509. PGP uses this as well as the notion of “trust.”

PGP provides a way for a public key to be “signed” by another public key. It also has a level of “trust” associated to each public key. The higher the level of trust, the stronger the binding of userid and key. A key which is signed by trusted keys is also trusted to a degree determined by number and degree of trust of the trusted keys.

The degrees of trust are: undefined, unknown user, usually not trusted to sign other keys, usually trusted to sign other keys, always trusted to sign other keys, and present in the secret key ring (ultimate trust).

If a user wishes to change one of his public keys or if he believes it has been compromised, then he widely disseminates a Key Revocation Certificate, signed by the associated private key.

Kerberos, developed by Project Athena at MIT, solves this problem: Assume an open distributed environment. Users at workstations wish to access services on various servers. Servers wish to restrict access to authorized users and be able to authenticate users requests for service. A workstation cannot be trusted to identify its users correctly.

There are three threats:

1. A user may gain access to a workstation and pretend to be another user on that workstation.

2. A user may alter the network address of a workstation so that the requests from it appear to come from a different workstation.

3. A user may eavesdrop on exchanges and use a replay attack to gain entrance to a server or to disrupt operations.

Kerberos addresses these threats by providing a central authentication server to authenticate users to servers and servers to users. It requires a user to prove identity for each service invoked. It also requires that servers prove their identity to clients.

Kerberos uses only conventional cryptography (DES), no public key crypto, and is supposed to be:

Secure: No eavesdropper can impersonate a user.

Reliable: No one can use any services unless permitted by Kerberos.

Transparent: User just types a password; all else is hidden.

Scalable: Can support many clients and servers.

Here is a simple authentication dialogue:

1. $C \rightarrow AS$: ID_C, P_C, ID_V

2. $AS \rightarrow C$: $Ticket$

3. $C \rightarrow V$: $ID_C, Ticket$

where $Ticket = E_{K_V}[ID_C, AD_C, ID_V]$.

AS = authentication server (Kerberos)

C = client

V = server

ID_C = identification of user on C

ID_V = identification of server V

P_C = password of user on C

AD_C = network address of C

K_V = secret key shared by AS and V.

The use of AD_C prevents ticket capture and reuse.

The ticket is valid only from workstation C.

The ticket is valid only once.

Some problems with this simple dialogue:

1. User on C must enter password for each ticket, too many times. Better to make ticket reusable.

2. The plaintext transmission of the password in Step 1. An eavesdropper could capture it.

Solve these problems by adding a TGS: Ticket granting server.

Once per user login session:

1. $C \to AS$: ID_C, ID_{TGS}

2. $AS \to C$: $E_{K_C}[Ticket_{TGS}]$,

where $Ticket_{TGS} =$
$= E_{K_{TGS}}[ID_C, AD_C, ID_{TGS}, TS_1, LT_1]$

Once per type of service (mail, print, login, etc.):

3. $C \to TGS$: $ID_C, ID_V, Ticket_{TGS}$

4. $TGS \to C$: $Ticket_V$

where $Ticket_V = E_{K_V}[ID_C, AD_C, ID_V, TS_2, LT_2]$

Once per service session:

5. $C \to V$: $ID_C, Ticket_V$

In 1 and 2, no password is sent over the network. Instead, in 2, C asks its user for a password (K_C) and uses it to decrypt the ticket.

The time stamps and lifetimes prevent reuse by an eavesdropper, unless he reuses it right away.

Two problems with the dialogue above:

(a) The lifetime may be too long or too short. TGS or V should be able to check that the person using the ticket is the same as the one to whom it was issued.

(b) Servers should have to prove their identity to users. Otherwise a bogus server could capture information from an unwary user and deny service.

These problems are solved by the Kerberos 4 dialogue on the next page.

Once per user login session:

1. $C \to AS$: ID_C, ID_{TGS}, TS_1

2. $AS \to C$:
$E_{K_C}[K_{C,TGS}, ID_{TGS}, TS_2, LT_2, Ticket_{TGS}]$,

where $Ticket_{TGS} =$
$E_{K_{TGS}}[K_{C,TGS}, ID_C, AD_C, ID_{TGS}, TS_2, LT_2]$

Once per type of service:

3. $C \to TGS$: $ID_V, Ticket_{TGS}, Authenticator_{C,TGS}$

where
$Authenticator_{C,TGS} = E_{K_{C,TGS}}[ID_C, AD_C, TS_3]$.

4. $TGS \to C$: $E_{K_{C,TGS}}[K_{C,V}, ID_V, TS_4, Ticket_V]$

where
$Ticket_V = E_{K_V}[K_{C,V}, ID_C, AD_C, ID_V, TS_4, LT_4]$.

Once per service session:

5. $C \to V$: $Ticket_V, Authenticator_{C,V}$

where $Authenticator_{C,V} = E_{K_{C,V}}[ID_C, AD_C, TS_5]$

6. $V \to C$: $E_{K_{C,V}}[TS_5 + 1]$

Now the lifetime is less important and can be made long enough since knowledge of $K_{C,TGS}$ and $K_{C,V}$ prove the user is the grantee of the ticket.

In 6, V proves its identity to C.

Bryn Dole and Steve Lodin, students in this class a few years ago, broke Kerberos 4.