Basic Cryptographic Protocols

Sources:

- *Applied Cryptography* by B. Schneier (primary source)
- *Making, Breaking Codes: An Introduction to Cryptology* by P. Garrett
- *Cryptography Decrypted* by H.X. Mel and D. Baker
- *Security in Computing* by C.P. Pfleeger
- *Cryptography and Network Security* by W. Stallings
- *Computer Security: Art and Science* by M. Bishop
- *Modern Cryptography* by W. Mao

Key-Exchange Protocols

These are used for distributing session keys.

We first consider a key-exchange protocol based on symmetric cryptography.

It assumes that the parties to communicate each share a secret key with a *Key Distribution Center* (*KDC*) [Schneier]:

1. Alice requests a session key from the KDC so she can communicate with Bob.

2. The KDC generates a random session key and encrypts two copies of it: one with Alice's key and the other with Bob's key. The KDC transmits both copies to Alice.

3. Alice decrypts her copy of the session key.

4. Alice sends Bob his copy of the session key.

5. Bob decrypts his copy of the session key.

6. Alice and Bob use this session key to communicate securely.

This protocol relies on the absolute security of the KDC, which is likely to be a computer program.

If the KDC is compromised, all network communications is compromised.

The KDS is also a single point of failure and a communications bottleneck.

The basic hybrid cryptosystem we saw earlier uses public-key cryptography to securely distribute symmetric session keys.

If signed public keys are available from a KDC database, the protocol becomes simpler [Schneier]:

1. Alice gets Bob's public key from the KDC.

2. Alice generates a random session key, encrypts it using Bob's public key, and sends it to Bob.

3. Bob then decrypts Alice's message using his private key.

4. Both of them encrypt their communications using the same session key.

Man-in-the-Middle Attack

In this attack, the attacker ("Mal") intercepts the communications of two parties and imitates each party when communicating with the other [Schneier]:

1. Alice sends Bob her public key. Mal intercepts this key and sends Bob his own public key.

2. Bob sends Alice his public key. Mal intercepts this key and sends Alice his own public key.

3. When Alice sends a message to Bob, Mal intercepts it, decrypts it with his private key, and re-encrypts it with Bob's public key.

4. When Bob sends a message to Alice, Mal intercepts it, decrypts it with his private key, re-encrypts it with Alice's public key, and sends it on to Alice.

This attack will also work if Alice's and Bob's keys are stored in a database.

The man-in-the-middle attack works because Alice and Bob cannot verify that they are talking to each other.

Data Integrity Techniques

A confidentiality service is inadequate for using a vulnerable communications channel securely.

Mechanisms are needed to enable a message receiver to verify that:

- A message has come from the claimed source
- A message has not been altered while in transit

Data integrity is the security service that protects against unauthorized modification of messages.

Rivest and Shamir's *interlock protocol* helps to prevent man-in-the middle attacks on full-duplex communication:

1. Alice sends Bob her public key.

2. Bob sends Alice his public key.

3. Alice encrypts her message using Bob's public key and transmits *half* of the encrypted message to Bob.

4. Bob encrypts his message using Alice's public key and transmits half of the encrypted message to Alice.

5. Alice transmits the other half of her encrypted message to Bob.

6. Bob puts the two halves of Alice's message together and decrypts it with his private key. Bob transmits the other half of his encrypted message to Alice.

7. Alice puts the two halves of Bob's message together and decrypts it with her private key.

Each party performs a step only after it has received the information sent by the other in the previous step.

The attacker, Mal, can still substitute his own public key for Alice's and Bob's.

However, when he intercepts half of Alice's message in step (3), he cannot decrypt it with his private key and re-encrypt it with Bob's public key.

He must invent a totally new message and transmit half of it to Bob.

He has the same problem when he intercepts half of Bob's message in step (4).

By the time Mal intercepts the second halves of the real messages in steps (5) and (6), it is too late for him to change the new messages he invented.

Thus, Mal's presence can be detected, although he does see Alice and Bob's messages.

Mal could possibly compromise this scheme if he knew the parties well enough to mimic both sides of a conversation between them.

Note that the interlock protocol cannot be effectively used to provide authentication.

(See S. M. Bellovin and M. Merritt. An Attack on the Interlock Protocol When Used for Authentication. *IEEE Transactions on Information Theory*, vol. 40, no. 1, January 1994, pp. 273-275.)

Key Exchange with Digital Signatures

Using digital signatures during a session-key exchange protocol also prevents this man-in-the-middle attack.

The KDC signs both parties' public keys.

The signed keys include a certification of ownership from the KDC.

When the parties receive the keys, they verify the KDC's signature.

This assures them that the public key belongs to the other person.

The key exchange protocol can then proceed as before.

An attacker cannot impersonate either party because he does not know either of their private keys.

He cannot substitute his public key for either of theirs.

The risk of compromising the KDC is less than with the protocol using symmetric cryptography.

If an attacker steals the KDC's private key, he can sign new keys but cannot decrypt session keys and read messages.

However he can create phony signed keys and impersonate users.

Key and Message Transmission

Messages can be exchanged before the key-exchange protocol is completed [Schneier]:

1. Alice generates a random session key *K* and encrypts message *M* using *K*.

   $E_K(M)$

2. Alice gets Bob's public key from the database.

3. Alice encrypts *K* with Bob's public key.

4. Alice sends both the encrypted message and the encrypted session key to Bob.

5. Bob decrypts Alice's session key, *K*, using his private key.

6. Bob decrypts Alice's message using the session key.

This is how public-key cryptography is most often used in communication.

Authentication

Authentication protocols are used to verify the identity of a user when they log onto a computer system.

Traditionally, passwords are used for authentication.

They can be guessed or stolen fairly easily in many cases.

More secure approaches to authentication involve cryptographic techniques.

Authentication With One-Way Functions

Needham and Guy recognized that a host does not need to store passwords.

It suffices to be able to recognize valid passwords.

This can be accomplished by storing only one-way functions of passwords [Schneier]:

1. Alice sends the host her password.

2. The host applies a one-way function to the password.

3. The host compares the result of the one-way function to the value it previously stored.

This protocol mitigates the threat of someone stealing the host's password list.

It is vulnerable to a *dictionary attack*:

1. The attacker compiles a large list of common passwords.

2. He applies the one-way function to each of them and stores the results.

3. He then steals the host's encrypted password list and looks for matches between its entries and the list he computed.

To make dictionary attacks impractical, "salt" can be used.

Salt is a random string that is concatenated with passwords before the one-way function is applied.

The salt value and the one-way function result are stored on a database on the host.

If the number of possible salt values is large enough, a dictionary attack is prevented.

This is because the attacker has to generate the one-way hash value for each possible salt value.

In effect, an attacker is forced to encrypt each password in his dictionary each time he wants to break someone's password.

It is important to use enough salt.

Klein developed a password-guessing program that often cracks 40% of passwords in a week on a UNIX host using 12 bits of salt.

Salt does not protect against an attack on a single password.

It also does not prevent a password from being read when it is transmitted to a host.

Authentication with Public-Key Cryptography

Public-key cryptography can be used to protect a password during authentication.

The host keeps a file with every user's public key.

Naive protocol [Schneier]:

1. The host sends Alice a random string.

2. Alice encrypts the string with her private key and sends it back to the host, along with her name.

3. The host looks up Alice's public key in its database and uses it to decrypt the message.

4. If the decrypted string matches the one sent to Alice, the host allows her access to the system.

This protocol is vulnerable to a chosen plaintext attack, which can reveal Alice's private key.

Secure authentication protocols are more complicated [Schneier]:

1. Alice performs a computation based on some random numbers and her private key and sends the result to the host.

2. The host sends Alice a different random number.

3. Alice does some computation based on the random numbers (both the ones she generated and the one she received from the host) and her private key, and she sends the result to the host.

4. The host does some computation on the various numbers received from Alice and her public key to verify that she knows her private key.

5. If she does, her identity is verified.

If Alice does not trust the host, Alice requires the host to prove its identity in the same way.

Authentication and Key Exchange

These protocols combine authentication with key exchange.

They allow two parties to exchange with confidence in each other's identities.

Symbols used in the protocols:

A	Alice's name
B	Bob's name
E_A	Encryption with a key that arbitrator shares with Alice
E_B	Encryption with a key that arbitrator shares with Bob
I	Index number
K	A random session key
L	Lifetime
T_A, T_B	A timestamp
R_A, R_B	A random number (*nonce*) chosen by Alice and Bob, respectively

Most of the protocols assume the arbitrator "Trent" shares a different secret key with each participant.

Wide-Mouth Frog Protocol

This is the simplest key management protocol using an arbitrator.

Both Alice and Bob share a secret key with the arbitrator, Trent.

It assumes Alice and Bob have clocks that are synchronized with Trent's clock.

Another strong assumption of the Wide-Mouth Frog protocol is that Alice is competent enough to generate good session keys.

Two messages are needed for Alice to transfer a session key to Bob [Schneier]:

1. Alice concatenates a timestamp, Bob's name, and a random session key and encrypts the whole message with the key she shares with Trent. She sends this to Trent, along with her name.

   $A, E_A(T_A, B, K)$

2. Trent decrypts the message from Alice. He checks that the timestamp is recent. If so, he then concatenates a new timestamp, Alice's name, and the random session key. He encrypts the whole message with the key he shares with Bob. Trent sends to Bob:

   $E_B(T_B, A, K)$

3. Bob checks that the timestamp is recent.

The timestamp checks should mean the session key is fresh, although Alice could have generated it long ago.

Kerberos

Kerberos is a key exchange and authentication protocol due to Denning and Sacco.

It is applicable to an enterprise environment, where a user is entitled to use a variety of distributed services, e.g.,

- A project server
- A Human Resources database
- An “intellectual property server”
- An “expenses server”

It is unreasonable to require a user to maintain several different cryptographic credentials.

Kerberos involves a *ticket-granting service* that issues a session key to be shared by a user and an application service *S*.

The ticket-granting server (Barnum) securely delivers this key inside a *ticket* encrypted with keys it shares with Alice and *S*.

Suppose Alice, wants to use an application server *S*.

Kerberos requires Alice to use two servers to obtain a credential that will authenticate her to *S*:

1. Alice must authenticate herself to the Kerberos system.

2. Then she must obtain a ticket from Barnum to use *S*.

The basis of Kerberos is the ticket credential, which contains:

$$Ticket_{Alice,Barnum} = \text{Barnum} \,||\, \{ \text{Alice} \,||\, \text{Alice address} \,||\, \text{valid time} \,||\, k_{Alice,Barnum} \} k_{Barnum}$$

k_{Barnum} is the key that Barnum shares with the authentication server.

$k_{Alice,Barnum}$ is the session key that Alice and Barnum will share.

The valid time is the time interval during which the ticket is valid (typically several minutes).

The ticket is the issuer's voucher for the identity of the requester of the service.

The *authenticator* contains the identity of the sender of the ticket.

It is used when Alice wants to show Barnum that the party sending the ticket is the same as the one to whom the ticket was issued.

It contains

$$A_{Alice,Barnum} = \{ \text{Alice} \,||\, \text{generation time} \,||\, kt \}\, k_{Alice,Barnum}$$

where $k_{Alice,Barnum}$ is the session key that Alice and Barnum share, *kt* is an alternative session key, and the authenticator was created at generation time.

Alice creates an authenticator whenever she sends a ticket. She sends both the ticket and the authenticator in the same message.

Alice wants to print using the service Guttenberg.

The authentication server is Cerberus.

The Kerberos Version 5 protocol:

1. Alice $\rightarrow$ Cerberus: Alice || Barnum

2. Cerberus $\rightarrow$ Alice: $\{ k_{Alice,Barnum} \} k_{Alice}$ || $Ticket_{Alice,Barnum}$

At this point, Alice decrypts the first part of the message to obtain they key she will use to communicate with Barnum.

3. Alice $\rightarrow$ Barnum: Guttenberg || $A_{Alice,Barnum}$ || $Ticket_{Alice,Barnum}$

4. Barnum $\rightarrow$ Alice: Alice || $\{ k_{Alice,Guttenberg} \} k_{Alice,Barnum}$ || $Ticket_{Alice,Guttenberg}$

5. Alice $\rightarrow$ Guttenberg: $A_{Alice,Guttenberg} \mid\mid Ticket_{Alice,Guttenberg}$

6. Guttenberg $\rightarrow$ Alice: $\{ t + 1 \} k_{Alice,Guttenberg}$

In these steps, Alice first constructs an authenticator and sends it, with the ticket and server name, to Barnum.

Barnum validates the request by comparing the data in the authenticator with the data in the ticket.

Because the ticket is encrypted using the key Barnum shares with Cerebus, he trusts it.

He then generates a session key and sends Alice a ticket to pass on to Guttenberg.

Step 5 is analogous to step 3.

Step 6 is optional; Alice may ask that Guttenberg send it to confirm the request.

Attacks Against Kerberos

If the clocks are not synchronized and if old tickets and authenticators are not cached, replay attacks are possible.

In Kerberos 5, authenticators are valid for 5 minutes, so tickets and authenticators can be replayed within that interval.

Because the tickets have some fixed fields, a dictionary attack can be used to determine keys shared by services or users and the ticket-granting service or the authentication service.

Researchers at Purdue reported that the session keys generated by Kerberos 4 were weak.