# Key Distribution and Authentication

When communicating, parties must use keys, and, hence, keys should be distributed before session begins. Also, parties should be sure that they communicate with the party which it claims to be. For example, am I sending my credit card number to Macy or to a Macy’s impersonator? Or, should the requestor be allowed to withdraw money from a Jody’s bank account?

# The Kerberos Protocol: Tickets

Kerberos system was designed at MIT. Kerberos is a middleware module that can be incorporated into a distributed computing system.

The Kerberos protocol involves the use of an intermediary process, called a key server (or, Key distribution server, KDS). KDS creates session keys on demand and distributes them in such a way that they are known only to communicating processes. For this reason, it is referred to as a trusted third party.

Each user wishing to participate in the protocol registers a symmetric user key with the KDS. User keys are not session keys. They are used only in the key distribution protocol at the start of a session.

Assume that a client, C, wants to communicate with a server, S. C and S have previously registered user keys  and , respectively, with KDS.  is known only to C and KDS. Similarly,  is known only to S and KDS. KDS is trusted by C in the sense that C assumes that KDS will never communicate  to any other process and that data structure it uses to keep  is protected from unauthorized access. Similarly, KDS is trusted by S.

Kerberos introduces the concept of a ticket to distribute a session key. To understand the role of a ticket, consider the following sequence of steps, illustrated in Figure 26.4, which forms the heart of the protocol.

# The Kerberos Protocol: Tickets (Cont 1)

Figure 26.4. Sequence of messages used to authenticate a client in symmetric encryption

# Protocol

1. C sends to KDS a message, M1 (in the clear), requesting a ticket to be used to authenticate C to S. M1 contains the names of the intended communicants (C,S).
2. When KDS receives M1, the following takes place:

(a) KDS (randomly) constructs a session key, Ksess,C&S

(b) KDS sends to C a message, M2, containing two items:

 (i) [Ksess,C&S, S, LT]

 (ii)  [Ksess,C&S, C, LT] – the actual ticket,

 where LT is the lifetime (the time interval) over which the ticket is valid.

1. When C receives M2, it performs the following steps:
	1. C recovers Ksess,C&S from the first item using  (it cannot decrypt the ticket)
	2. C saves the ticket until it is ready to request some service

Observe that KDS does not know the actual source of M1: it could have been sent by an intruder, I, posing as C. However, KDS encrypts M2,

making the information returned accessible only to C and S. Key  can be constructed by C using a one-way function of the user’s password PC:



# Protocol (Cont 1)

Thus, only C, using PC, can construct, and hence only C can retrieve Ksess,C&S from the first item in M2.

Later, when C wants to request service from S, the following takes place:

1. C sends to S a message, M3, containing the arguments of the request (they might or might not be encrypted using Ksess,C&S), the ticket, and an authenticator (see below).

Only S can decrypt the ticket and recover the items it contains. However, the ticket (containing C) alone is not sufficient to authenticate C to S since I could have copied it in step 2 and replayed it to S with its own request. A timestamp might be useful in preventing a replay, but the timestamp cannot be stored in the ticket because the ticket is meant to be used by C multiple times during its lifetime. C therefore sends an authenticator along with its ticket. An authenticator consists of C’s name together with a current timestamp, TS, encrypted with Ksess,C&S:



and is meant to be used only once. S can decrypt the authenticator by using Ksess,C&S (which it determines by decrypting the ticket).

At this point, S knows that the ticket could have been constructed only by KDS since only KDS knows KS,KDS. Furthermore, since S trusts KDS and since each time Ksess,C&S is transmitted it is encrypted by either KC,KDS or KS,KDS, S knows that only C (and KDS) knows Ksess,C&S. The authenticator contains some plaintext (e.g., C) encrypted by Ksess,C&S, that can be compared with the contents of the ticket (which also contains C). If they match, S concludes that C must have constructed the authenticator.

However, several possible attacks must be ruled out:

1. I attempts a replay attack in which it copies both the ticket and the authenticator from M3 and uses them at a later time. To combat this, we must take it impossible for an authenticator (in contrast to a ticket) to be used more than once. A new authenticator (with a unique timestamp) is constructed by C for each of its requests. The authenticator is live if its time timestamp is within the lifetime (LT) of the accompanying ticket. To ensure that a copy of an authenticator is of no value, and that S can defend itself against a replay, S uses the following protocol:
	1. If the received authenticator is not live, S rejects it
	2. S maintains a list of authenticators it has received that are still live. If the received authenticator is live, S compares it against the list and rejects it if a copy is found. By maintaining lifetime information, S can limit the number of authenticators it has keep on the list

# Protocol (Cont 2)

1. I intercepts M3 (it does not reach S) and tries to use the ticket and authenticator for its own request for service. However, if C has chosen to encrypt the arguments of its request with Ksess,C&S, then I cannot substitute its own arguments because he does not know Ksess,C&S. Sending the entire intercepted message at a later time accomplishes nothing for I since it simply causes C’s original request to be serviced.
2. I intercepts M1 (it does not reach KDS) and substitutes the message (C,I). KDS responds to C with a ticket encrypted with. I’s goal in this attack is to copy the message M3 that it hopes that C will subsequently send to S. Since the ticket is encrypted with I’s private key, I can extract the session key. In this case S will not be able to decrypt the ticket, but I can determine private information about C contained in the arguments that C sends to S. The protocol defends against this attack by including the server’s name in the first item of M2, which in this case will be I instead of S. C uses this information to determine the identity of the process that can decrypt its request message.

# Single Sign-On

Kerberos provides a property, referred to as single sign-on, that is becoming important as client interactions become increasingly complex. Complex interactions frequently involve access to multiple resources and hence multiple servers. Each server needs to authenticate the client and, in the worst case, has its own interface for doing so. If the client uses the same password for all servers, security can be compromised; if the client uses different passwords, he must remember all of them. In either case he must engage in multiple authentication protocols, and the system administrator must keep the authentication information associated with each server current as client information changes.

With single sign-on, the client needs to authenticate itself only once. Kerberos provides this property by concentrating authentication in an authentication server, AS, which authenticates C at login time using the password supplied by the client. Since the identity of the servers the client intends to access might be not known at this time, it is not possible for AS to construct the appropriate tickets (since each ticket must be encrypted with a particular server’s key). Instead, it returns a ticket-granting ticket to C, which is used for requesting service from a particular server, called the ticket-granting server, TGS – also part of Kerberos. Later C can request the specific ticket it needs (for example, a ticket for S) from TGS using the ticket-granting ticket.

The authentication server generates a session key, , that C can use to communicate with TGS, and returns to C (in format similar to that of M2 in the simplified protocol):

* -  is a session key for communicating with TGS
*  - the ticket-granting ticket for TGS

where  and  are keys that C and TGS have registered with AS.

Later, when C wants to access a particular server, S, it sends a copy of the ticket-granting ticket together with the server’s name (and an authenticator) to TGS. TGS then returns to C (again in a format similar to that of M2):

*  -  is a session key for communicating with S
*  - this is a ticket for S (note that S’s private key,  is available to TGS).

C thus obtains a different ticket for each server it accesses. It engages in a single authentication protocol, and since the use of tickets is not visible at the user level, the user interface is simplified. Also, since authentication is concentrated in a single server, the administration of authentication information is simplified.

# Nonces

Suppose two processes, P1 and P2, share a session key, , and P1 sends an encrypted message, M1, to P2 and expects an encrypted reply, M2. When P1 receives M2, how can P1 be sure that it was constructed by P2? It might seem that P1 can just decrypt M2 using  and see if the result makes sense. Often, however, determining whether a string makes sense requires human intervention, and in some cases even that does not help. Consider the case in which M2 simply contains a data string (an arbitrary string of bits – perhaps the weight of some device) calculated by P2. An intruder might substitute a random string for M2. When P1 decrypts that string using , it might produce another string that looks like a data string. Unfortunately, P1 cannot determine whether or not the string is correct without repeating P2’s calculation (which it is not in a position to do). Alternatively, the intruder might replay an earlier message sent during the same session and hence encrypted with . In some cases, such a replay might be a possible correct response to M1 (two devices could have the same weight), and hence P1 is fooled into accepting it.

A nonce can be used to solve this problem. A nonce is a bit string created by one process in a way that makes it highly unlikely that another process can create the same string. For example, a randomly created bit string of sufficient length created in one process probably will not be created later by another process. Nonces have a variety of use, one of which is related to authentication.

To solve the above problem, P1 include a nonce, N, in M1, and P2 includes N+1 in M2. On receipt of M2, P1 knows that the sender must have decrypted M1, since N+1, not a simple replay of N, is returned. This implies that the sender knows  and is therefore P2.

In Kerberos, the timestamp TS (which is already part of the authenticator) can be used as a nonce so that no additional items need be added. The server can include TS+1 in M4.

Nonces are often used in cryptographic protocols for a completely different reason. Appending a large random number to the plaintext before encrypting a message makes it considerably harder for an intruder to decrypt the message by guessing parts of its contents – for example, guessing the expiration time or some other information in a credit card – and using that information to reduce the cost of a brute-force search to discover the key. This use of a nonce is sometimes referred to as adding salt to a message. In some protocols, a nonce used for this purpose is called a confounder.