Documentation
MIT Kerberos Consortium
Publications
Tutorial
Useful Links

Kerberos is an authentication protocol for trusted hosts on untrusted networks.

KERBEROS PROTOCOL TUTORIAL

This tutorial was written by Fulvio Ricciardi and is reprinted here with his permission. Mr. Ricciardi works at the National Institute of Nuclear Physics in Lecce, Italy. He is also the author of the Linux project zeroshell.net, where he originally published this tutorial. Thank you, Mr. Ricciardi!

Document version: 1.0.3 (11/27/2007)

Author: Fulvio Ricciardi (Fulvio.Ricciardi@le.infn.it)
INFN - the National Institute of Nuclear Physics
Computing and Network Services - LECCE, Italy

1 Introduction
2 Aims
3 Definitions of components and terms
3.1 Realm
3.2 Principal
3.3 Ticket
3.4 Encryption
3.4.1 Encryption type
3.4.2 Encryption key
3.4.3 Salt
3.4.4 Key Version Number (kvno)
3.5 Key Distribution Center (KDC)
3.5.1 Database
3.5.2 Authentication Server (AS)
3.5.3 Ticket Granting Server (TGS)
3.6 Session Key
3.7 Authenticator
3.8 Replay Cache
3.9 Credential Cache

4 Kerberos Operation
4.1 Authentication Server Request (AS_REQ)
4.2 Authentication Server Reply (AS_REP)
4.3 Ticket Granting Server Request (TGS_REQ)
4.4 Ticket Granting Server Replay (TGS_REP)
4.5 Application Server Request (AP_REQ)
4.6 Pre-Authentication [4.6 Application Server Replay (AP_REP) missing]
5 Tickets in-depth
5.1 Initial tickets
5.2 Renewable tickets
5.3 Forwardable tickets
6 Cross Authentication
6.1 Direct trust relationships
6.2 Transitive trust relationships
6.3 Hierarchical trust relationships

1 Introduction

The Kerberos protocol is designed to provide reliable authentication over open and insecure networks where communications between the hosts belonging to it may be intercepted. However, one should be aware that Kerberos does not provide any guarantees if the computers being used are vulnerable: the authentication servers, application servers (imap, pop, smtp, telnet, ftp, ssh , AFS, lpr, ...) and clients must be kept constantly updated so that the authenticity of the requesting users and service providers can be guaranteed.

The above points justify the sentence: **"Kerberos is an authentication protocol for trusted hosts on untrusted networks"**. By way of example, and to reiterate the concept: Kerberos' strategies are useless if someone who obtains privileged access to a server, can copy the file containing the secret key. Indeed, the intruder will put this key on another machine, and will only have to obtain a simple spoof DNS or IP address for that server to appear to clients as the authentic server.

2 Aims

Before describing the elements that make up the Kerberos authentication system and looking at its operation, some of the aims the protocol wishes to achieve are listed below:

- The user's password must never travel over the network;

- The user's password must never be stored in any form on the client machine: it must be immediately discarded after being used;
- The user's password should never be stored in an unencrypted form even in the authentication server database;
- The user is asked to enter a password only once per work session. Therefore users can transparently access all the services they are authorized for without having to re-enter the password during this session. This characteristic is known as **Single Sign-On**;
- Authentication information management is centralized and resides on the authentication server. The application servers must not contain the authentication information for their users. This is essential for obtaining the following results:
    1. The administrator can disable the account of any user by acting in a single location without having to act on the several application servers providing the various services;
    2. When a user changes its password, it is changed for all services at the same time;
    3. There is no redundancy of authentication information which would otherwise have to be safeguarded in various places;
- Not only do the users have to demonstrate that they are who they say, but, when requested, the application servers must prove their authenticity to the client as well. This characteristic is known as **Mutual authentication**;
- Following the completion of authentication and authorization, the client and server must be able to establish an encrypted connection, if required. For this purpose, Kerberos provides support for the generation and exchange of an encryption key to be used to encrypt data.

3 Definition of the components and terms

This section provides the definition of the objects and terms, knowledge of which is essential for the subsequent description of the Kerberos protocol. Since many definitions are based on others, wherever possible I have tried to put them in order so that the meaning of a term is not given before defining it. However, it may be necessary to read this section twice to fully understand all the terms.

3.1 Realm

The term realm indicates an authentication administrative domain. Its intention is to establish the boundaries within which an authentication server has the authority to authenticate a user, host or service. This does not mean that the authentication between a user and a service that they must belong to the same realm: if the two objects are part of different realms and there is a trust relationship between them, then the authentication can take place. This characteristic, known as **Cross-Authentication** will be described below.

Basically, a user/service belongs to a realm if and only if he/it shares a secret (password/key) with the authentication server of that realm.

The name of a realm is case sensitive, i.e. there is a difference between upper and lower case letters, but normally realms always appear in upper case letters. It is also good practice, in an organization, to make the realm name the same as the DNS domain (in upper case letters though). Following these tips when selecting the realm name significantly simplifies the configuration of Kerberos clients, above all when it is desired to establish trust relationships with subdomains. By way of example, if an organization belongs to the DNS domain example.com, it is appropriate that the related Kerberos realm is EXAMPLE.COM.

3.2 Principal

A principal is the name used to refer to the entries in the authentication server database. A principal is associated with each user, host or service of a given realm. A principal in Kerberos 5 is of the following type:

component1/component2/.../componentN@REALM

However, in practice a maximum of two components are used. For an entry referring to a user the principal is the following type:

Name[/Instance]@REALM

The instance is optional and is normally used to better qualify the type of user. For example administrator users normally have the admin instance. The following are examples of principals referred to users:

pippo@EXAMPLE.COM admin/admin@EXAMPLE.COM pluto/admin@EXAMPLE.COM

If, instead, the entries refer to services, the principals assume the following form:

Service/Hostname@REALM

The first component is the name of the service, for example imap, AFS, ftp. Often it is the word host which is used to indicate generic access to the machine (telnet, rsh, ssh). The second component is the complete hostname (FQDN) of the machine providing the requested service. It is important that this component exactly matches (in lower case letters) the DNS reverse resolution of the application server's IP address. The following are valid examples of principals referring to services:

imap/mbox.example.com@EXAMPLE.COM
host/server.example.com@EXAMPLE.COM
afs/example.com@EXAMPLE.COM

It should be noted that the last case is an exception because the second component is not a hostname but the name of the AFS cell that the principal refers to. Lastly, there are principals which do not refer to users or services but play a role in the operation of the authentication system. An overall example is krbtgt/REALM@REALM with its associated key is used to encrypt the Ticket Granting Ticket (we'll look at this later).

In Kerberos 4 there can never be more than two components and they are separated by the character "." instead of "/" while the hostname in the principals referring to services is the short one, i.e. not the FQDN. The following are valid examples:

pippo@EXAMPLE.COM pluto.admin@EXAMPLE.COM imap.mbox@EXAMPLE.COM

3.3 Ticket

A ticket is something a client presents to an application server to demonstrate the authenticity of its identity. Tickets are issued by the authentication server and are encrypted using the secret key of the service they are intended for. Since this

key is a secret shared only between the authentication server and the server providing the service, not even the client which requested the ticket can know it or change its contents. The main information contained in a ticket includes:

- The requesting user's principal (generally the username);
- The principal of the service it is intended for;
- The IP address of the client machine from which the ticket can be used. In Kerberos 5 this field is optional and may also be multiple in order to be able to run clients under NAT or multihomed.
- The date and time (in timestamp format) when the tickets validity commences;
- The ticket's maximum lifetime
- The session key (this has a fundamental role which is described below);

Each ticket has an expiration (generally 10 hours). This is essential since the authentication server no longer has any control over an already issued ticket. Even though the realm administrator can prevent the issuing of new tickets for a certain user at any time, it cannot prevent users from using the tickets they already possess. This is the reason for limiting the lifetime of the tickets in order to limit any abuse over time.

Tickets contain a lot of other information and flags which characterize their behavior, but we won't go into that here. We'll discuss tickets and flags again after seeing how the authentication system works.

3.4 Encryption

As you can see Kerberos often needs to encrypt and decrypt the messages (tickets and authenticators) passing between the various participants in the authentication. It is important to note that Kerberos uses only symmetrical key encryption (in other words the same key is used to encrypt and decrypt). Certain projects (e.g. pkinit) are active for introducing a public key system in order to obtain the initial user authentication through the presentation of a private key corresponding to a certified public key, but since there is no standard we'll skip this discussion for now.

3.4.1 Encryption type

Kerberos 4 implements a single type of encryption which is DES at 56 bits. The weakness of this encryption plus other protocol vulnerabilities have made Kerberos 4 obsolete. Version 5 of Kerberos, however, does not predetermine the number or type of encryption methodologies supported. It is the task of each specific implementation to support and best negotiate the various types of encryption. However, this flexibility and expandability of the protocol has accentuated interoperability problems between the various implementations of Kerberos 5. In order for clients and application and authentication servers using different implementations to interoperate, they must have at least one encryption type in common. The difficulty related to the interoperability between Unix implementations of Kerberos 5 and the one present in the Active Directory of Windows is a classic example of this. Indeed, Windows Active Directory supports a limited number of encryptions and only had DES at 56 bits in common with Unix. This required keeping the latter enabled, despite the risks being well known, if interoperability had to be guaranteed. The problem was subsequently solved with version 1.3 of MIT Kerberos 5. This version introduced RC4-HMAC support, which is also present in Windows and is more secure than DES. Among the supported encryptions (but not by Windows) the triple DES (3DES) and newer AES128 and AES256 are worth mentioning.

3.4.2 Encryption key

As stated above, one of the aims of the Kerberos protocol is to prevent the user's password from being stored in its unencrypted form, even in the authentication server database. Considering that each encryption algorithm uses its own key length, it is clear that, if the user is not to be forced to use a different password of a fixed size for each encryption method supported, the encryption keys cannot be the passwords. For these reasons the **string2key** function has been introduced, which transforms an unencrypted password into an encryption key suitable for the type of encryption to be used. This function is called each time a user changes password or enters it for authentication. The string2key is called a hash function, meaning that it is irreversible: given that an encryption key cannot determine the password which generated it (unless by brute force). Famous hashing algorithms are MD5 and CRC32.

3.4.3 Salt

In Kerberos 5, unlike version 4, the concept of password salt has been introduced. This is a string to be concatenated to the unencrypted password before applying the string2key function to obtain the key. Kerberos 5 uses the same principal of the user as salt:

$$K_{pippo} = string2key (P_{pippo} + "pippo@EXAMPLE.COM")$$

K_{pippo} is the encryption key of the user pippo and P_{pippo} is the unencrypted password of the user.

This type of salt has the following advantages:

- Two principals belonging to the same realm and having the same unencrypted password, still have different keys. For example, imagine an administrator having a principal for everyday work (pippo@EXAMPLE.COM) and one for administrative work (pippo/admin@EXAMPLE.COM). It is very likely that this user has set the same password for both principals for reasons of convenience. The presence of the salt guarantees that the related keys are different.
- If a user has two accounts in different realms, it is fairly frequent that the unencrypted password is the same for both realms: thanks to the presence of the salt, a possible compromise of an account in one realm, will not automatically cause the other to be compromised.

A null salt can be configured for compatibility with Kerberos 4. Vice versa, for compatibility with AFS, it is possible to configure a salt which is not the complete name of the principal, but simply the name of the cell.

Having discussed the concepts of encryption type, string2key and salt, it is possible to check the accuracy of the following observation: in order that there is interoperability between the various Kerberos implementations, it is not sufficient to negotiate a common type of encryption, but the same types of string2key and salt need to be used as well.
It is also important to note that, in explaining the concepts of string2key and salt, reference was only made to the user principals and never to those of servers. The reason is clear: a service, even if it shares a secret with the authentication server, is not an unencrypted password (who would enter it?), but a key which, once generated by the administrator on the Kerberos server, is memorized as it is on the server providing the service.

3.4.4 Key Version Number (kvno)

When a user changes a password or an administrator updates the secret key for an application server, this change is logged by advancing a counter. The current value of the counter identifying the key version, is known as the Key Version Number or more briefly **kvno**.

3.5 Key Distribution Center (KDC)

We have spoken generically about the authentication server. Since it is the fundamental object involved in the authentication of users and services, we will now take a more in-depth look at it without going into all the details of its operation, which are instead the topic of the section dedicated to protocol operation.
The authentication server in a Kerberos environment, based on its ticket distribution function for access to the services, is called Key Distribution Center or more briefly KDC. Since it resides entirely on a single physical server (it often coincides with a single process) it can be logically considered divided into three parts: Database, Authentication Server (AS) and Ticket Granting Server (TGS). Let's take a brief look at them.

Note: it is possible to make the server redundant within the realm in a Master/Slave (MIT and Heimdal) or Multimaster structure (Windows Active Directory). How to obtain redundancy is not described by the protocol, but depends on the implementation being used and will not be discussed here.

3.5.1 Database

The database is the container for entries associated with users and services. We refer to an entry by using the principal (i.e. the name of the entry) even if often the term principal is used as a synonym for entry. Each entry contains the following information:

- The principal to which the entry is associated;
- The encryption key and related kvno;
- The maximum validity duration for a ticket associated to the principal;
- The maximum time a ticket associated to the principal may be renewed (only Kerberos 5);
- The attributes or flags characterizing the behavior of the tickets;
- The password expiration date;
- The expiration date of the principal, after which no tickets will be issued.

In order to make it more difficult to steal the keys present in the database, the implementations encrypt the database using the **master key**, which is associated with the principal K/M@REALM. Even any database dumps, used as backups or for propagation from the KDC master towards the slave, are encrypted using this key, which it is necessary to know in order to reload them.

3.5.2 Authentication Server (AS)

The Authentication Server is the part of the KDC which replies to the initial authentication request from the client, when the user, not yet authenticated, must enter the password. In response to an authentication request, the AS issues a special ticket known as the Ticket Granting Ticket, or more briefly TGT, the principal associated with which is

krbtgt/REALM@REALM. If the users are actually who they say they are (and we'll see later how they demonstrate this) they can use the TGT to obtain other service tickets, without having to re-enter their password.

3.5.3 Ticket Granting Server (TGS)

The Ticket Granting Server is the KDC component which distributes service tickets to clients with a valid TGT, guaranteeing the authenticity of the identity for obtaining the requested resource on the application servers. The TGS can be considered as an application server (given that to access it it is necessary to present the TGT) which provides the issuing of service tickets as a service. It is important not to confuse the abbreviations TGT and TGS: the first indicates a ticket and the second a service.

3.6 Session Key

As we have seen, the users and services share a secret with the KDC. For users, this secret is the key derived from their password, while for services, it is their secret key (set by the administrator). These keys are called long term, since they do not change when the work session changes.

However, it is necessary that the user also shares a secret with the service, at least for the time in which a client has a work session open on a server: this key, generated by the KDC when a ticket is issued, is called the Session Key. The copy intended for the service is enveloped by the KDC in the ticket (in any case their application server knows the long term key and can decode it and extract the session key), while the copy intended for the user is encapsulated in an encrypted packet with the user long term key. The session key plays a fundamental role in demonstrating the authenticity of the user which we will see in the following paragraph.

3.7 Authenticator

Even if the user principal is present in a ticket and only the application server can extract and possibly manage such information (since the ticket is encrypted with the secret key of the service), this is not enough to guarantee the authenticity of the client. An impostor could capture (remember the hypothesis of an open and insecure network) the ticket when it is sent by a legitimate client to the application server, and at an opportune time, send it to illegitimately obtain the service. On the other hand, including the IP addresses of the machine from where it is possible to use it is not very useful: it is known that in an open and insecure network addresses are easily falsified. To solve the problem, one has to exploit the fact that the client and server, at least during a session have the session key in common that only they know (also the KDC knows it since it generated it, but it is trusted by definition!!!). Thus the following strategy is applied: along with the request containing the ticket, the client adds another packet (the authenticator) where the user principal and time stamp (its at that time) are included and encrypts it with the session key; the server which must offer the service, upon receiving this request, unpacks the first ticket, extracts the session key and, if the user is actually who he/she says, the server is able to unencrypt the authenticator extracting the timestamp. If the latter differs from the server time by less than 2 minutes (but the tolerance can be configured) then the authentication is successful. This underlines the criticality of synchronization between machines belonging to the same realm.

3.8 Replay Cache

The possibility exists for an impostor to simultaneously steal both the ticket and the authenticator and use them during the 2 minutes the authenticator is valid. This is very difficult but not impossible. To solve this problem with Kerberos 5, Replay Cache has been introduced. In application servers (but also in TGS), there exists the capacity to remember authenticators which have arrived within the last 2 minutes, and to reject them if they are replicas. With this the problem is resolved as long as the impostor is not smart enough to copy the ticket and authenticator and make them arrive at the application server before the legitimate request arrives. This really would be a hoax, since the authentic user would be rejected while the impostor would have access to the service.

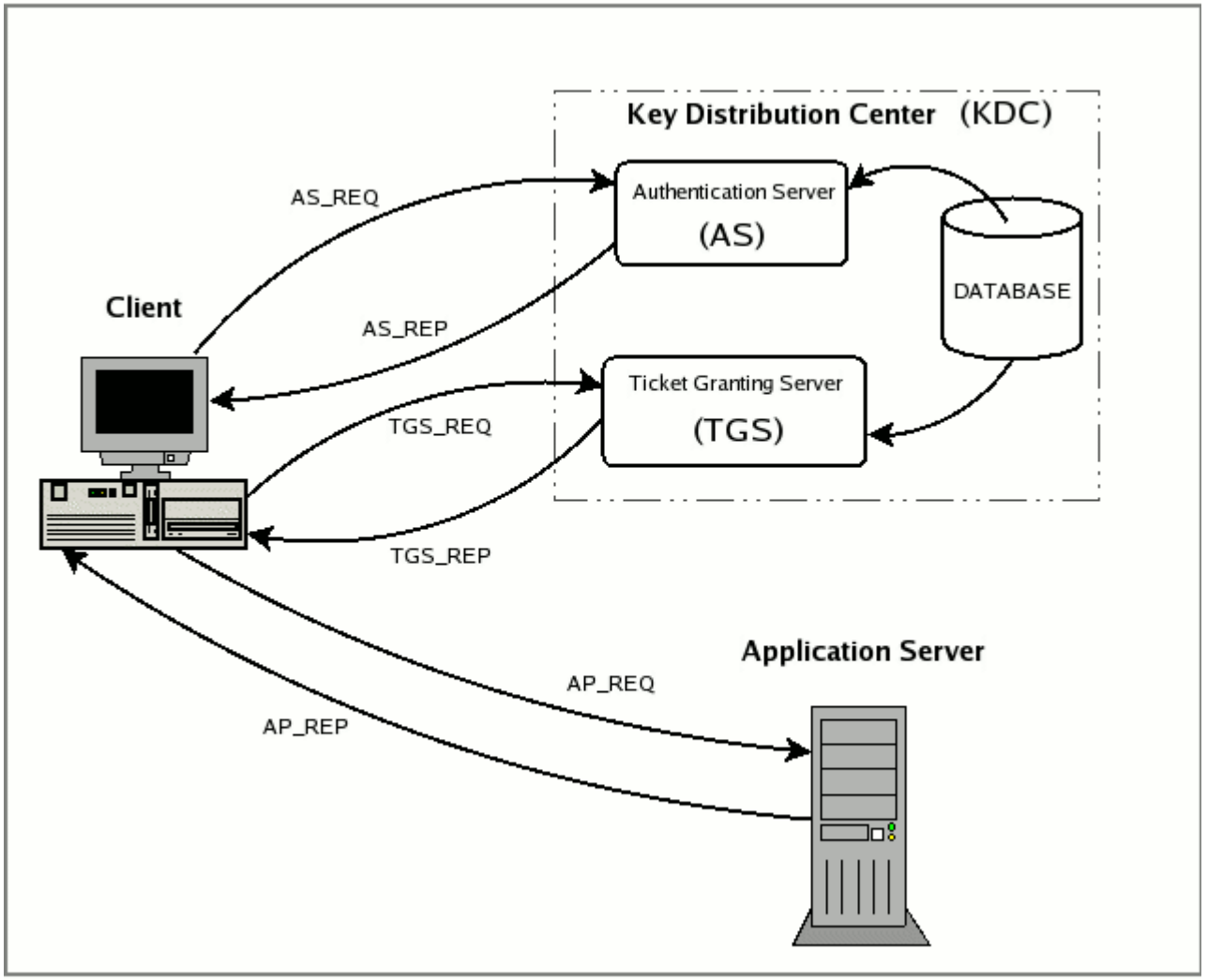
3.9 Credential Cache

The client never keeps the user's password, nor does it memorise the secret key obtained by applying string2key: they are used to decrypt the replies from KDC and immediately discarded. However, on the other hand, to implement the single sign-on (SSO) characteristic, where the user is asked to enter the password just once per work session, it is necessary to memorise the tickets and related session key. The place where this data is stored is called the "Credential Cache". Where this cache needs to be located does not depend on the protocol, but varies from one implementation to another. Often for portability purposes they are located in the filesystem (MIT and Heimdal). In other implementations (AFS and Active Directory), in order to increase security in the event of vulnerable clients, the credential cache is placed in an area of the memory accessible only to kernels and not swappable on the disk.

4 Kerberos Operation

Finally, having acquired the concepts described in the preceding paragraphs, it is possible to discuss how Kerberos operates. We'll do this by listing and describing each of the packets which go between the client and KDC and between client and application server during authentication. At this point, it is important to underline that an application server never communicates directly with the Key Distribution Center: the service tickets, even if packeted by TGS, reach the service only through the client wishing to access them. The messages we will discuss are listed below (see also the figure below):

- **AS_REQ** is the initial user authentication request (i.e. made with kinit) This message is directed to the KDC component known as Authentication Server (AS);
- **AS_REP** is the reply of the Authentication Server to the previous request. Basically it contains the TGT (encrypted using the TGS secret key) and the session key (encrypted using the secret key of the requesting user);
- **TGS_REQ** is the request from the client to the Ticket Granting Server (TGS) for a service ticket. This packet includes the TGT obtained from the previous message and an authenticator generated by the client and encrypted with the session key;
- **TGS_REP** is the reply of the Ticket Granting Server to the previous request. Located inside is the requested service ticket (encrypted with the secret key of the service) and a service session key generated by TGS and encrypted using the previous session key generated by the AS;
- **AP_REQ** is the request that the client sends to an application server to access a service. The components are the service ticket obtained from TGS with the previous reply and an authenticator again generated by the client, but this time encrypted using the service session key (generated by TGS);
- **AP_REP** is the reply that the application server gives to the client to prove it really is the server the client is expecting. This packet is not always requested. The client requests the server for it only when mutual authentication is necessary.



Now each of the previous phases is described in greater detail with reference to Kerberos 5, but pointing out the differences with version 4. Nevertheless, it should be borne in mind that the Kerberos protocol is rather complicated and this document is not intended as a guide for those who wish to know the exact operating details (in any case, these are already written up in RFC1510). The discussion below has been left intentionally abstract, but sufficient for those who examine the KDC logs to understand the various authentication transitions and any problems which occur.

Note: the subsequent paragraphs enclose unencrypted data in round brackets (), and encrypted data in curly brackets {}: (x, y, z) means that x, y, z are unencrypted; { x, y, z }K indicates that x, y, z are encrypted all together using the symmetrical key K. It is also important to note that the order in which the components are listed in a packet has nothing to do with the real order found in the actual messages (UDP or TCP). This discussion is very abstract. Should you wish further details, please refer to RFC1510 having a good background on the descriptive protocol ASN.1

4.1 Authentication Server Request (AS_REQ)

In this phase, known as the initial authentication request, the client (kinit) asks the KDC (more specifically the AS) for a Ticket Granting Ticket. The request is completely unencrypted and looks like this:

$$AS\_REQ = (Principal_{Client} , Principal_{Service} , IP\_list , Lifetime)$$

where: $Principal_{Client}$ is the principal associated with the user seeking authentication (e.g. pippo@EXAMPLE.COM); $Principal_{Service}$ is the principal associated to the service the ticket is being asked for and thus is the string "krbtgt/REALM@REALM" (see note*); `IP_list` is a list of IP addresses that indicate the host where it is possible to use the ticket which will be issued (see note **); `Lifetime` is the maximum validity time (requested) for the ticket to be issued.

Note *: it may seem superfluous to add the $Principal_{service}$ to the initial authentication request since this would be constantly set to the TGS principal i.e. krbtgt/REALM@REALM. However, this is not the case, indeed, a user planning to use just one service during a work session, would not use the Single Sign-on, and may ask the AS directly for the ticket for this service, thus skipping the subsequent request to the TGS. From an operational standpoint (MIT 1.3.6) the following command is sufficient:* `kinit -S imap/mbox.example.com@EXAMPLE.COM pippo@EXAMPLE.COM`

Note *: `IP_list` may also be null. In this case the corresponding ticket can be used by any machine. This resolves the problem of those clients under NAT, since their request would arrive at the service with a source address different from that of the requesting user, but the same as the router making the NAT. Instead, in the case of machines with more than one network card, `IP_list` should contain the IP addresses of all the cards: in fact it would be difficult to predict beforehand with which connection the server which provides the service would be contacted.*

4.2 Authentication Server Reply (AS_REP)

When the previous request arrives, the AS checks whether $Principal_{Client}$ and $Principal_{Service}$ exist in the KDC database: if at least one of the two does not exist an error message is sent to the client, otherwise the Authentication Server processes the reply as follows:

- It randomly creates a session key which will be the secret shared between the client and the TGS. Let's say SK_{TGS};
- It creates the Ticket Granting Ticket putting it inside the requesting user's principal, the service principal (it is generally krbtgt/REALM@REALM, but read the note* for the previous paragraph), the IP address list (these first three pieces of information are copied as they arrive by the AS_REQ packet), date and time (of the KDC) in timestamp format, lifetime (see note*) and lastly the session key. SK_{TGS}; the Ticket Granting Ticket thus appears as follows:

$$TGT = (Principal_{Client} , krbtgt/REALM@REALM , IP\_list , Timestamp , Lifetime , SK_{TGS})$$

- It generates and sends the reply containing: the ticket created previously, encrypted using the secret key for the service (let's call it K_{TGS}); the service principal, timestamp, lifetime and session key all encrypted using the secret key for the user requesting the service (let's call it K_{User}). In summary:

$$AS\_REP = \{ Principal_{Service} , Timestamp , Lifetime , SK_{TGS} \}K_{User} \{ TGT \}K_{TGS}$$

It may seem that this message contains redundant information ($Principal_{Service}$, timestamp, lifetime and session key). But this is not the case: since the information present in the TGT is encrypted using the secret key for the server, it cannot be read by the client and needs to be repeated. At this point, when the client receives the reply message, it will ask the user to enter the password. The salt is concatenated with the password and then the string2key function is applied: with the resulting key an attempt is made to decrypt the part of the message encrypted by the KDC using the secret key of the user stored in the database. If the user is really who he/she says, and has thus entered the correct password, the decrypting operation will be successful and thus the session key can be extracted and with the TGT (which remains encrypted) stored in the user's credential cache.

Note *: *the actual lifetime, i.e. which is put in the ticket is the lowest of the following values: the lifetime requested by the client, the one contained in the user's principal and that in the service principal. Actually in terms of implementation another limit can be set from the configuration of the KDC and applied to any ticket.*

4.3 Ticket Granting Server Request (TGS_REQ)

At this point, the user who has already proved to be who he/she says (thus in his/her credential cache there is a TGT and session key SK_{TGS} and wants to access the service but does not yet have a suitable ticket, sends a request (TGS_REQ) to the Ticket Granting Service constructing it as follows:

- Create an authenticator with the user principal, client machine timestamp and encrypt everything with the session key shared with the TGS, i.e.:

$$Authenticator = \{ Principal_{Client} , Timestamp \}SK_{TGS}$$

- Create a request packet containing: the service principal for which the ticket is needed and lifetime uncrypted; the Ticket Granting Ticket which is already encrypted with the key of the TGS; and the authenticator just created. In summary:

$$TGS\_REQ = (Principal_{Service} , Lifetime , Authenticator) \{ TGT \}K_{TGS}$$

4.4 Ticket Granting Server Replay (TGS_REP)

When the previous request arrives, the TGS first verifies that the principal of the requested service ($Principal_{Service}$) exists in the KDC database: If it exists, it opens the TGT using the key for krbtgt/REAM@REALM and extracts the session key (SK_{TGS}) which it uses to decrypt the authenticator. For the service ticket to be issued it checks that the following conditions have a positive outcome:

- the TGT has not expired;
- The $Principal_{Client}$ present in the authenticator matches the one present in the TGT;
- The authenticator is not present in the replay cache and has not expired;
- If `IP_list` is not null it checks that the source IP address of the request packet (TGS_REQ) is one of those contained in the list;

The previously checked conditions prove that the TGT really belongs to the user who made the request and therefore the TGS starts to process the reply as follows:

- It randomly creates a session key which will be the secret shared between the client and the service. Let's say $SK_{Service}$;
- It creates the service ticket, putting inside the requesting user's principal, the service principal, the list of IP addresses, the date and time (of the KDC) in timestamp format, the lifetime (as the minimum between the lifetime of the TGT and that associated with the service principal) and lastly the session key $SK_{Service}$. Known as $T_{Service}$ the new ticket is:

$$T_{Service} = (Principal_{Client} , Principal_{Service} , IP\_list , Timestamp , Lifetime , SK_{Service})$$

- It sends the reply message containing: the previously created ticket, encrypted using the service secret key (let's call it $K_{Service}$); the service principal, timestamp, lifetime and new session key all encrypted using the session key extracted from TGT. In summary:

$$TGS\_REP = \{ Principal_{Service} , Timestamp , Lifetime , SK_{Service} \}SK_{TGS} \{ T_{Service} \}K_{Service}$$

When the client receives the reply, having in the credential cache the session key SK_{TGS}, it can decrypt the part of the message containing the other session key and memorise it together with the service ticket $T_{Service}$ which, however, remains encrypted.

4.5 Application Request (AP_REQ)

The client, having the credentials to access the service (i.e. the ticket and related session key), can ask the application server for access to the resource via an AP_REQ message. It should be borne in mind that, unlike the previous messages where the KDC was involved, the AP_REQ is not standard, but varies depending on the application. Thus, the application programmer has the job of establishing the strategy with which the client will use its credentials to prove its identity to the server. However, we can consider the following strategy by way of example:

- The client creates an authenticator containing the user principal and timestamp and encrypts everything with the session key $SK_{Service}$ that it shares with the application server, i.e.:

$$Authenticator = \{ Principal_{Client} , Timestamp \}SK_{Service}$$

- It creates a request packet containing the service ticket $T_{Service}$ which is encrypted with its secret key and the authenticator just created. In summary:

$$AP\_REQ = Authenticator \{ T_{Service} \}K_{Service}$$

When the previous request arrives, the application server opens the ticket using the secret key for the requested service and extracts the session key $SK_{Service}$ which it uses to decrypt the authenticator. To establish that the requesting user is authentic and thus grant access to the service, the server verifies the following conditions:

- the ticket has not expired;
- The $Principal_{Client}$ present in the authenticator matches the one present in the ticket;
- The authenticator is not present in the replay cache and has not expired;
- If `IP_list` (extracted from the ticket) is not null it checks that the source IP address of the request packet (AP_REQ) is one of those contained in the list;

***Note**: the previous strategy is very similar to the one used by the Ticket Granting Server to check the authenticity of the user requesting a service ticket. But this should not be surprising, since we have already explained that the TGS can be considered as an application server whose service is to provide tickets to those who prove their identity with a TGT.*

4.6 Pre-Authentication

As seen in the description of the Authentication Server Reply (AS_REP), before distributing a ticket the KDC simply checks that the principal of the requesting user and service provider exist in the database. Then, particularly if it involves a request for a TGT, it is even easier, because krbtgt/REALM@REALM certainly exists and thus it is sufficient to know that a user's principal exists to be able to obtain a TGT with a simple initial authentication request. Obviously, this TGT, if the request comes from an illegitimate user, cannot be used because they do not know the password and cannot obtain the session key for creating a valid authenticator. However, this ticket, obtained in such a easy way can undergo a brute-force attack in an attempt to guess the long-term key for the service the ticket is intended for. Obviously, guessing the secret of a service is not any easy thing even with current processing powers, however, with Kerberos 5, a pre-authentication concept has been introduced to reinforce security. Thus if the KDC policies (configurable) request pre-authentication for an initial client request, the Authentication Server replies with an error packet indicating the need to pre-authenticate. The client, in light of the error, asks the user to enter the password and resubmit the request but this time adding the timestamp encrypted with the user long term key, which, as we know, is obtained by applying the string2key to the unencrypted password after having added the salt, if there is one. This time, the KDC, since it knows the secret key of the user, attempts to decrypt the timestamp present in the request and if it is successful and the timestamp is in line, i.e. included within the established tolerance, it decides that the requesting user is authentic and the authentication process continues normally.
It is important to note that pre-authentication is a KDC policy and thus the protocol does not necessarily require it. In terms of implementation, MIT Kerberos 5 and Heimdal have pre-authentication disabled by default, while Kerberos within Windows Active Directory and the AFS kaserver (which is a pre-authenticated Kerberos 4) request it.

5 Tickets in further detail

As mentioned earlier, now that the operation of the KDC and messages between the hosts involved in the authentication have been discussed, we can now turn to the tickets. These, depending on whether they have attributes (also called flags) set inside them, behave in a certain manner. I have listed the most important types of tickets below, and even if not completely correct given that I are talking about a protocol, I will refer (just enough to make things clear) to version 1.3.6 of MIT Kerberos 5.

5.1 Initial tickets

An initial ticket is what is obtained directly from AS, i.e. when users must authenticate by entering the password. From here, it may be deduced that the TGT is always an initial ticket. On the other hand, the service tickets are distributed by the TGS upon presentation of a TGT and thus are not initial tickets. However, there is an exception to this rule: in order to guarantee that the user entered the password only a few seconds before, some Kerberos applications may request that the service ticket be initial; in this case the ticket, despite not being a TGT, is requested from the AS instead of the TGS and is thus an initial ticket. In operational terms, the user pippo, wishing to obtain a ticket which is initial (thus without using the TGT) for an imap service on the machine mbox.example.com uses the command:

```
[pippo@client01 pippo]$ kinit -S imap/mbox.example.com@EXAMPLE.COM pippo@EXAMPLE.COM
Password for pippo@EXAMPLE.COM:
[pippo@client01 pippo]$
[pippo@client01 pippo]$ klist -f
```

```
Ticket cache: FILE:/tmp/krb5cc_500
Default principal: pippo@EXAMPLE.COM

Valid starting     Expires            Service principal
01/27/05 14:28:59  01/28/05 14:28:39  imap/mbox.example.com@EXAMPLE.COM
        Flags: I

Kerberos 4 ticket cache: /tmp/tkt500
klist: You have no tickets cached
```

The presence of the flag I should be noted, indicating that it is an initial ticket.

5.2 Renewable tickets

A renewable ticket can be resubmitted to the KDC for renewal, i.e. it is reassigned the entire lifetime. Obviously, the KDC will honour the renewal request only if the ticket has not expired yet and has not exceeded the maximum renewal time (set in the Key Distribution Center database). Being able to renew a ticket combines the necessity of having short duration tickets for security reasons, with not having to re-enter the password for long periods: for example, imagine a job which must be processed for days and must not require any human intervention. In the following example pippo asks for a ticket which lasts for a maximum of one hour but is renewable for 8 days:

```
kinit -l 1h -r 8d pippo
Password for pippo@EXAMPLE.COM:
[pippo@client01 pippo]$
[pippo@client01 pippo]$ klist -f
Ticket cache: FILE:/tmp/krb5cc_500
Default principal: pippo@EXAMPLE.COM

Valid starting     Expires            Service principal
01/27/05 15:35:14  01/27/05 16:34:54  krbtgt/EXAMPLE.COM@EXAMPLE.COM
        renew until 02/03/05 15:35:14, Flags: RI


Kerberos 4 ticket cache: /tmp/tkt500
klist: You have no tickets cached
```

while for pippo to renew his ticket without re-entering the password:

```
[pippo@client01 pippo]$ kinit -R
[pippo@client01 pippo]$
[pippo@client01 pippo]$ klist -f
Ticket cache: FILE:/tmp/krb5cc_500
Default principal: pippo@EXAMPLE.COM

Valid starting     Expires            Service principal
01/27/05 15:47:52  01/27/05 16:47:32  krbtgt/EXAMPLE.COM@EXAMPLE.COM
        renew until 02/03/05 15:35:14, Flags: RIT
```

```
Kerberos 4 ticket cache: /tmp/tkt500
klist: You have no tickets cached
```

5.3 Forwardable tickets

Let's suppose we have a work session on a machine with the related TGT and wish to login from it onto another machine, keeping the ticket. Forwardable tickets are the solution to this problem. A ticket forwarded from one host to another is in itself forwardable, thus once authenticated it is possible to access the login on all the desired machines without having to re-enter any password.
To obtain the same result without Kerberos, it would be necessary to use much less secure methods such as rsh or public key authentication with ssh. However, this latter method may be impracticable on systems where the user home directories are on network filesystems (e.g. NFS or AFS) since the private key (which should be private) would go over the network.

6 Cross Authentication

We've already mentioned the possibility for a user belonging to a certain realm to authenticate and access the services of another realm. This characteristic known as cross-authentication is based on the assumption that there is a trust relationship between the realms involved. This may be mono-directional, meaning that the users of realm A can access the services of realm B but not vice versa, or bi-directional, where, as one might expect, the opposite is also possible. In the following paragraphs we will look at cross authentication, breaking down the trust relationships into direct, transitive and hierarchical.

6.1 Direct trust relationships

This type of trust relationship is elementary and is the basis of cross-authentication and is used to construct the other two types of relationships we will look at later. It occurs when the KDC of realm B has direct trust in the KDC of realm A, thus allowing the users of the latter realm to access its resources. From a practical point of view, a direct trust relationship is obtained by having the two involved KDCs share a key (the keys become two if a bi-directional trust is desired). To do this the concept of a remote Ticket Granting Ticket is introduced which, in the example of the two realms A and B, assumes the form krbtgt/B@A and is added to both the KDCs with the same key. This key is the secret which will guarantee the trust between the two realms. Obviously, to make it bi-directional (i.e. that A also trusts B), it is necessary to create the remote TGT krbtgt/A@B in both KDCs, associating them with another secret key.

As we'll see shortly in the following example, the introduction of the remote TGTs makes cross authentication a natural generalization of normal intra-realm authentication: this underlines that the previous description of Kerberos operation continues to be valid as long as it is accepted that the TGS of one realm can validate the remote TGTs issued by the TGS of another. Note the formal anomaly arising when the remote TGTs are not issued by the AS, as happens for the local ones, but by the local Ticket Granting Server upon presentation of the local TGT.
Now let's look at an example to clarify all this. Let's suppose that the user pippo of the realm EXAMPLE.COM, whose

associated principal is pippo@EXAMPLE.COM, wishes to access the pluto.test.com server belonging to the TEST.COM realm, via ssh:

- If Pippo does not already have a TGT in the realm EXAMPLE.COM he makes an initial authentication request (kinit). Obviously, the reply comes from the AS of his realm;
- He gives the `ssh pippo@pluto.test.com` command which should open the remote shell on pluto.test.com without having to re-enter the password;
- the ssh client makes two queries to DNS: it works out the IP of pluto.test.com and on the just obtained address carries out the reverse in order to obtain the hostname (FQDN) in canonical form (in this case it coincides with pluto.test.com);
- ssh client then realizes, thanks to the previous result, that the destination does not belong to the user's realm and thus asks the TGS of the realm EXAMPLE.COM (note that it asks the TGS of its realm for this) for the remote TGT krbtgt/TEST.COM@EXAMPLE.COM;
- With the remote TGT it asks the TGS of the realm TEST.COM for the host/pluto.test.com@TEST.COM service ticket;
- When the TEST.COM Ticket Granting Service receives the request, it checks for the existence of the principal krbtgt/TEST.COM@EXAMPLE.COM in its database with which it can verify the trust relationship. If this verification is positive the service ticket (encrypted with the key of host/pluto.test.com@TEST.COM) is finally issued which pippo will send to the host pluto.test.com to obtain the remote shell.

6.2 Transitive trust relationships

When the number of realms in which cross-authentication must be possible increases, the number of keys to exchange increases quadratically. For example, if there are 5 realms and the relationships must be bi-directional, the administrators must generate 20 keys (double the combinations of 5 elements by 2 by 2).

To get around this problem, Kerberos 5 has introduced transitivity in the trust relationship: if realm A trusts realm B and realm B trusts realm C then A will automatically trust C. This relationship property drastically reduces the number of keys (even if the number of authentication passages increases).

However, there is still a problem: the clients cannot guess the authentication path (capath) if it is not direct. So they must be informed of the correct path by creating a special stanza ([capaths]) in the configuration of each of the clients. These paths must also be known to the KDCs which will use them to check the transits.

6.3 Hierarchical trust relationships

If, within organizations, the convention of naming realms with the name of DNS domains in upper case letters is used (highly recommended choice) and if the latter belong to a hierarchy, then Kerberos 5 will support adjacent realms (hierarchically) having a trust relationship (naturally this assumed trust must be supported by the presence of appropriate keys) and will automatically construct (without the need for capaths) the transitive authentication paths. However, administrators can alter this automatic mechanism (for example for reasons of efficiency) by forcing the capaths in the client configuration.