Explain like I’m 5: Kerberos

roguelynn.com/words/explain-like-im-5-kerberos

April 2, 2013

Apr 2, 2013
Explain like I’m 5 years old: Kerberos – what is Kerberos, and why should I care?

While this topic probably can not be explained to a 5 year-old and be understood, this is my attempt at defragmenting documentation with some visual aids and digestible language.

In a nutshell

Basically, Kerberos comes down to just this:

- a protocol for authentication
- uses tickets to authenticate
- avoids storing passwords locally or sending them over the internet
- involves a trusted 3rd-party
- built on symmetric-key cryptography

You have a **ticket** – your proof of identity encrypted with a secret key for the particular service requested – on your local machine (creation of a ticket is described below); so long as it’s valid, you can access the requested service that is within a Kerberos realm.

Typically, this is used within corporate/internal environments. Perhaps you want to access your internal payroll site to review what little bonus your boss has given you. Rather than re-entering your user/password credentials, your ticket (cached on your system) is used to authenticate allowing for single sign-on.

Your ticket is refreshed when you sign on to your computer, or when you `kinit USER` within your terminal.

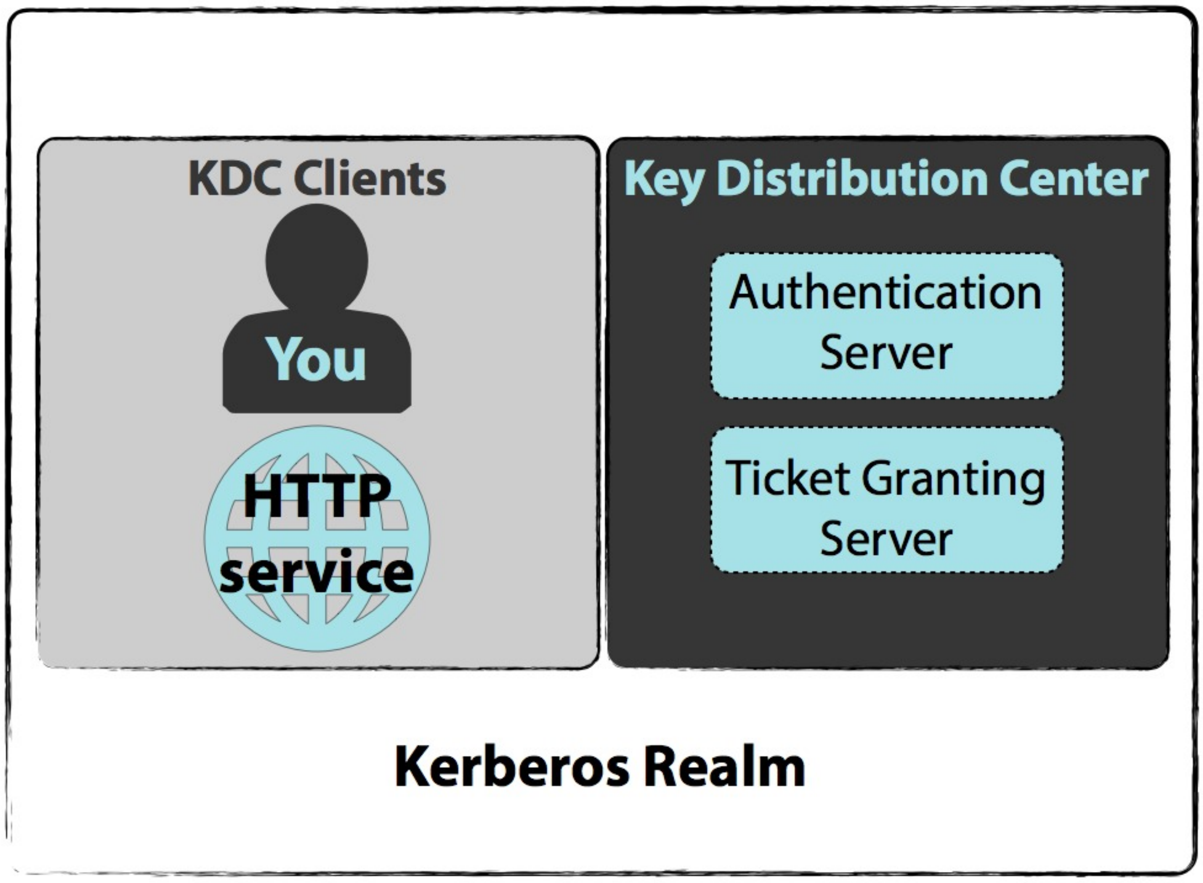
For the trivia-loving folks, Kerberos’ name comes from Greek mythology, the three-headed guard dog of Hades. It’s pretty fitting since it takes a third-party (a Key Distribution Center) to authenticate between a client and a service or host machine.

Wikipedia

Kerberos Realm

Admins create realms – Kerberos realms – that will encompass all that is available to access. Granted, **you** may not have access to certain services or host machines that is defined within the policy management – developers should not access anything finance related, stuff like that. But a realm defines what Kerberos manages in terms of who can access what.

Your machine, the Client, lives within this realm, as well as the service or host you want to request and the Key Distribution Center, KDC (no, not the KGB, although I always think of that, too). In the following example, I separate out the Authentication Server and the Ticket Granting Server, but both are within the KDC.



To keep in the back of your mind

You may want to come back up here after you read through the gritty details on how the example works.

When requesting access to a service or host, three interactions take place between you and:

- the Authentication Server
- the Ticket Granting Server
- the Service or host machine that you're wanting access to.

Other important points:

- With each interaction, you'll receive two messages. Each message is one that you can decrypt, and one that you can not.
- The service or machine you are requesting access to **never** communicates directly with the KDC.
- The KDC stores all of the secret keys for user machines and services in its database.
- Secret keys are passwords plus a salt that are hashed – the hash algorithm is chosen during implementation of the Kerberos setup. For services or host machines, there are no passwords (who would enter it). A key is actually generated by an admin during initial setup and memorized on the service/host machine.

- **Again**, these secret keys are all stored in the KDC database; recall the Kerberos' reliance on symmetric-key cryptography.
- The KDC itself is encrypted with a master key to add a layer of difficulty from stealing keys from the database.
- There are Kerberos configurations and implementations that use public-key cryptography instead of symmetrical key encryption.

An aside: the order of the messages and their contents discussed here does not reflect the order in which they are sent over TCP or UDP.

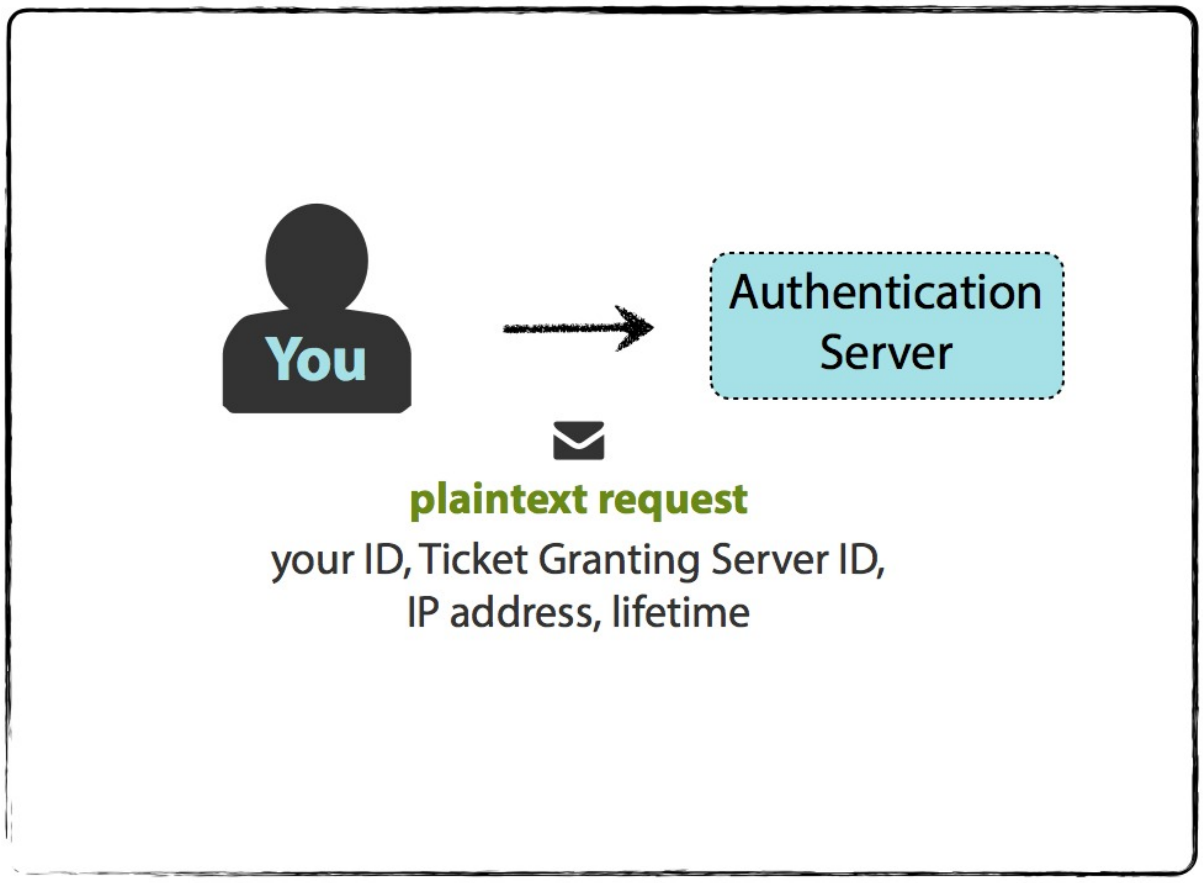
The example below describes what happens when you request something from an internal HTTP Service – like information regarding payroll within your corporate intranet.

You and the Authentication Server

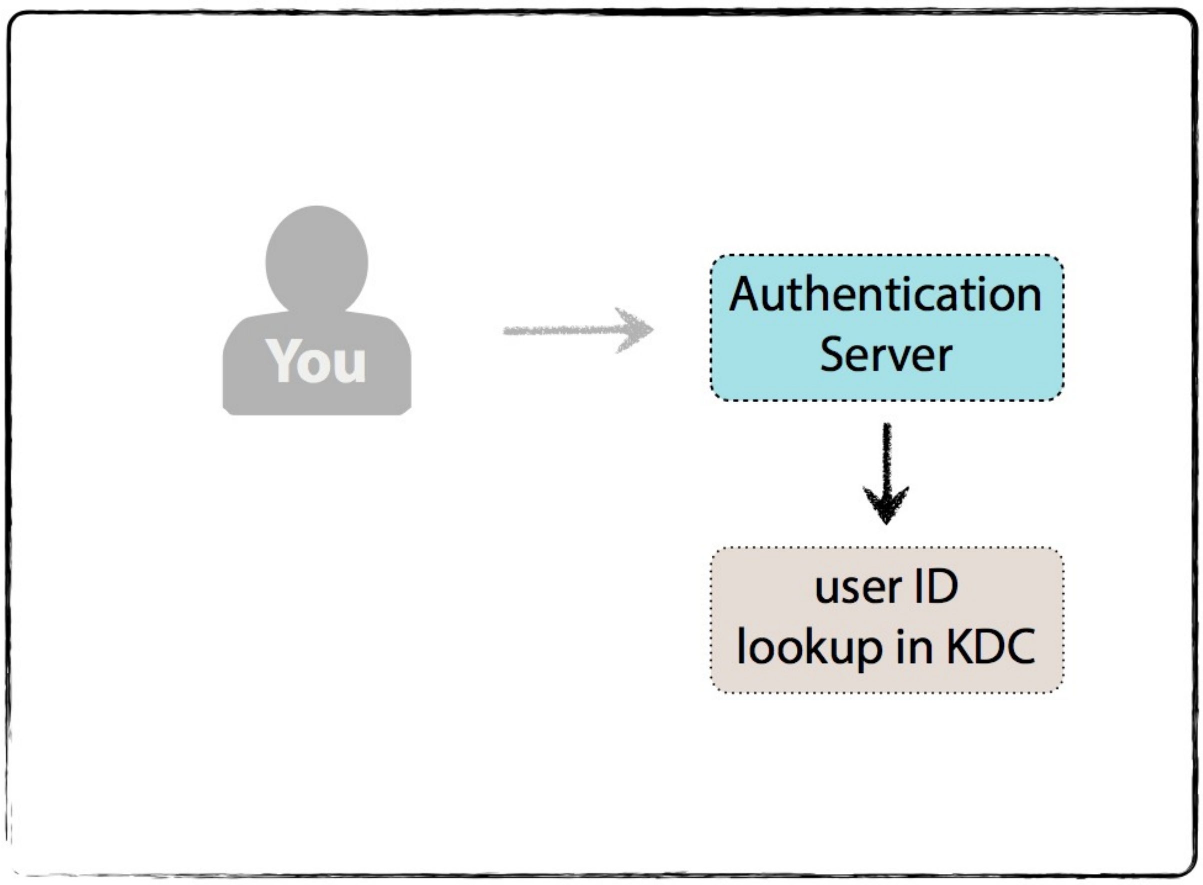
You want to access an HTTP Service, but first you must introduce yourself to the Authentication Server. Logging into your computer, or `kinit USERNAME`, initiates that introduction via a plaintext request for a Ticket Granting Ticket (TGT). The plaintext message contains:

- your name/ID
- the name/ID of the requested service (in this case, service is the Ticket Granting Server),
- your network address (may be a list of IP addresses for multiple machines, or may be null if wanting to use on any machine), and
- requested lifetime for the validity of the TGT,

and is sent to the Authentication Server.



The Authentication Server will check if you are in the KDC database. This check is only to see if you exist; no credentials are checked.



If there are no errors (e.g. user is not found), it will randomly generate a key called a session key for use between you and the Ticket Granting Server (TGS).

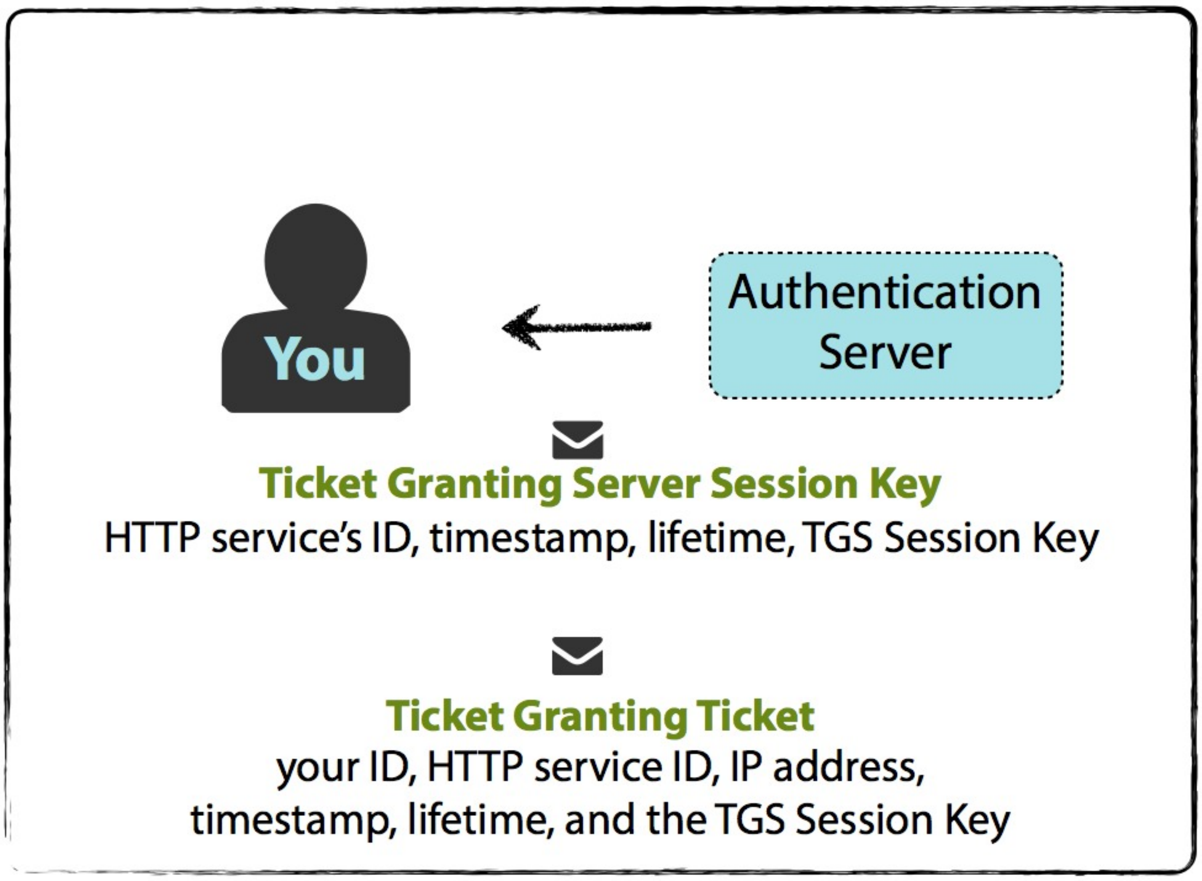
The Authentication Server will then send two messages back to you. One message is the TGT that contains:

- your name/ID,
- the TGS name/ID,
- timestamp,
- your network address (may be a list of IP addresses for multiple machines, or may be null if wanting to use on any machine)
- lifetime of the TGT (could be what you initially requested, lower if you or the TGS's secret keys are about to expire, or another limit that was implemented during the Kerberos setup), and
- TGS Session Key,

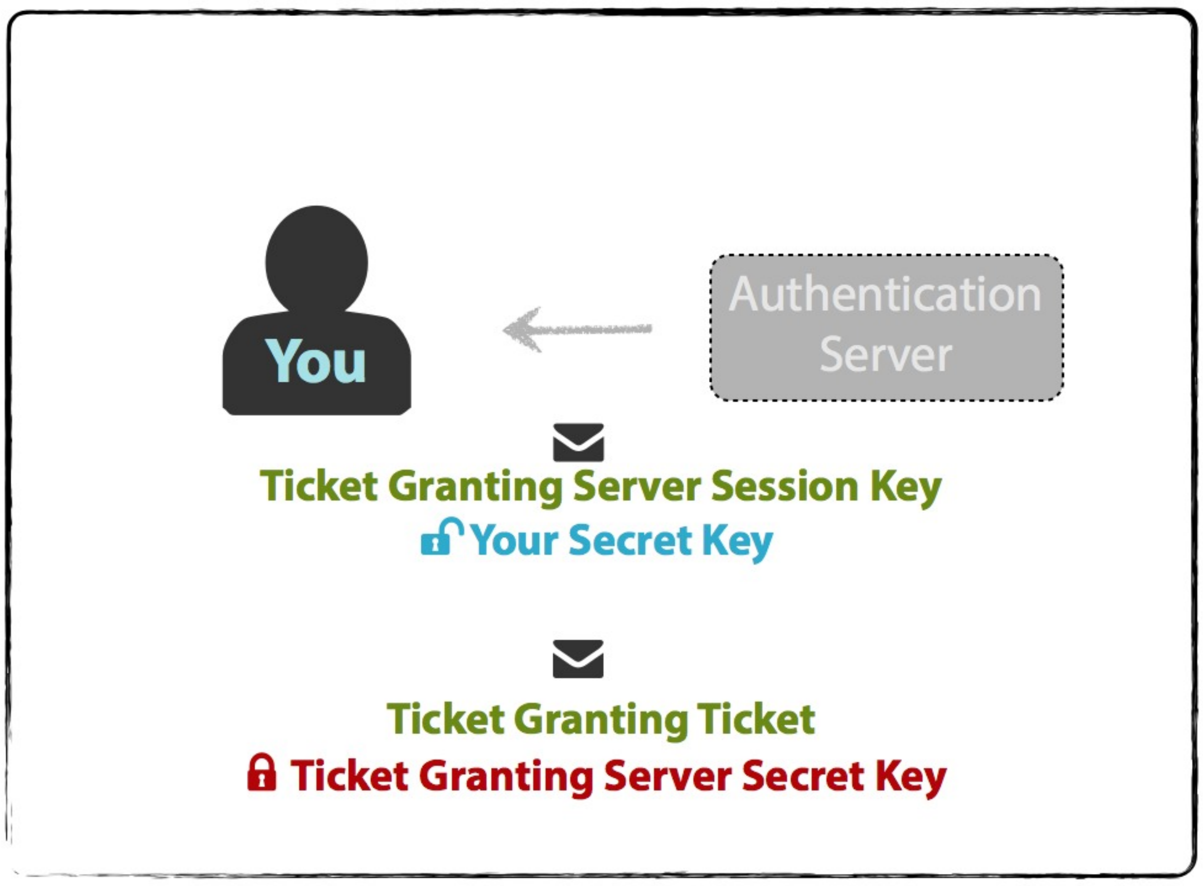
and is encrypted with the TGS Secret Key . The other message contains:

- the TGS name/ID,
- timestamp,
- lifetime (same as above), and
- TGS Session Key

and is encrypted with your Client Secret Key. Note that the TGS Session Key is the shared key between you and the TGS.


You
Authentication Server
Ticket Granting Server Session Key
HTTP service's ID, timestamp, lifetime, TGS Session Key
Ticket Granting Ticket
your ID, HTTP service ID, IP address, timestamp, lifetime, and the TGS Session Key

Your Client Secret Key is determined by prompting you for your password, appending a salt (made up of `user@REALMNAME.COM`) and hashing the whole thing. Now you can use it for decrypting the second message in order to obtain the TGS Session Key. If the password is incorrect, then you will not be able to decrypt the message. Please note that this is the step in which the password you enter is implicitly validated.


You
Authentication Server
Ticket Granting Server Session Key
Your Secret Key
Ticket Granting Ticket
Ticket Granting Server Secret Key

You can not, however, decrypt the TGT since you do not know the TGS Secret Key. The encrypted TGT is stored within your credential cache.

You and the Ticket Granting Server

At this point, you have the TGT that you can not read because you do not have the TGS Secret Key to decrypt it. You do, however, have the TGS Session Key.

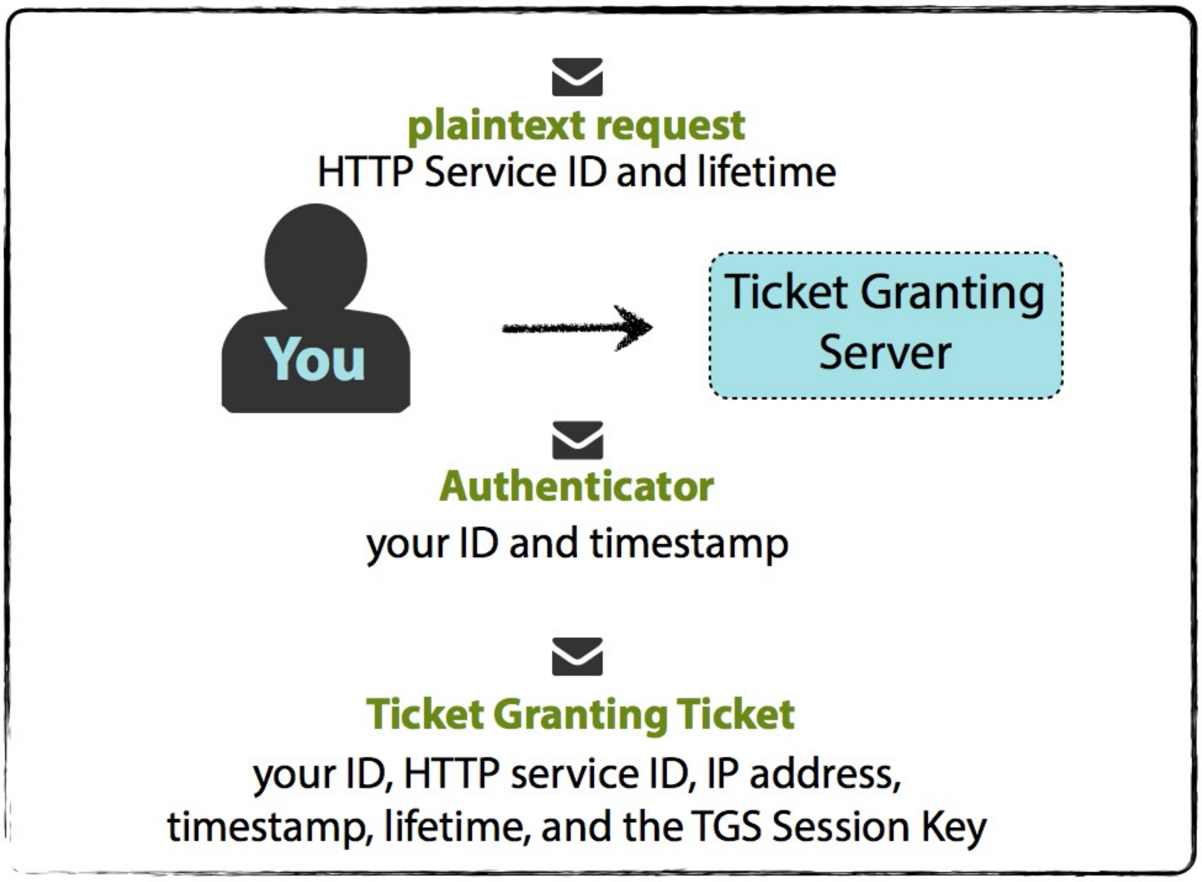
It's now your turn to send two messages. You first prepare the Authenticator, encrypted with the TGS Session Key, containing:

- your name/ID, and
- timestamp.

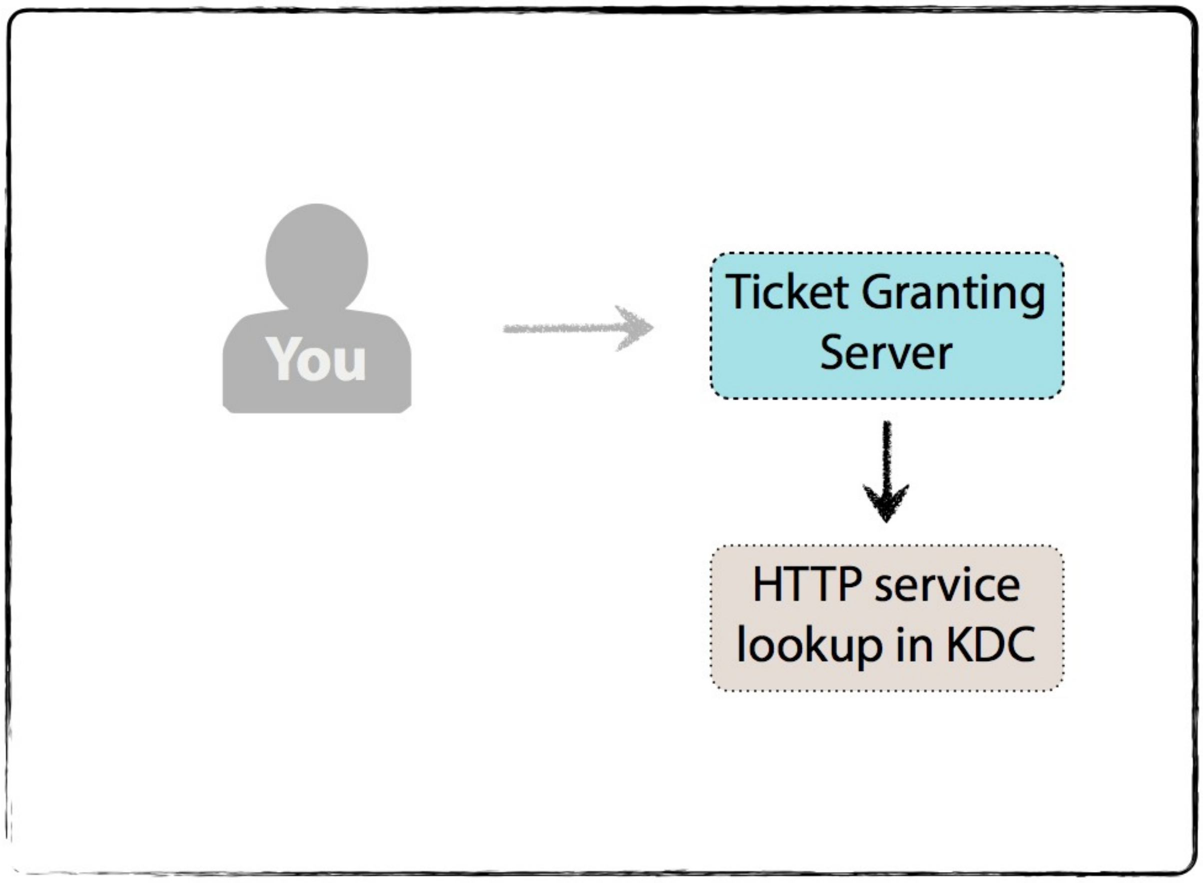
You send an unencrypted message that contains:

- the requested HTTP Service name/ID you want access to, and
- lifetime of the Ticket for the HTTP Service,

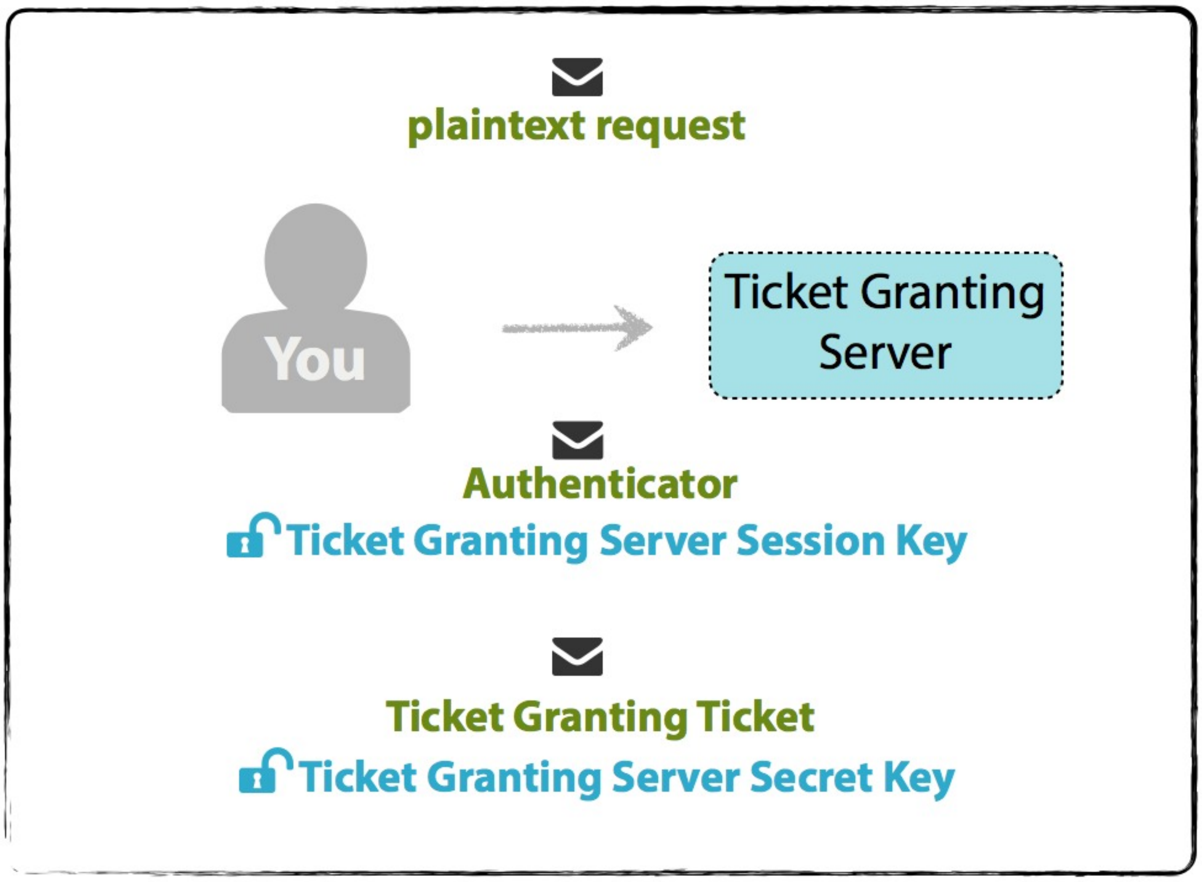
along with the encrypted Authenticator and TGT to the Ticket Granting Server.


plaintext request
HTTP Service ID and lifetime
You
Ticket Granting Server
Authenticator
your ID and timestamp
Ticket Granting Ticket
your ID, HTTP service ID, IP address, timestamp, lifetime, and the TGS Session Key

The Ticket Granting Server will first check the KDC database to see if the HTTP Service exists.



If so, the TGS decrypts the TGT with its Secret Key . Since the now-unencrypted TGT contains the TGS Session Key, the TGS can decrypt the Authenticator you sent.



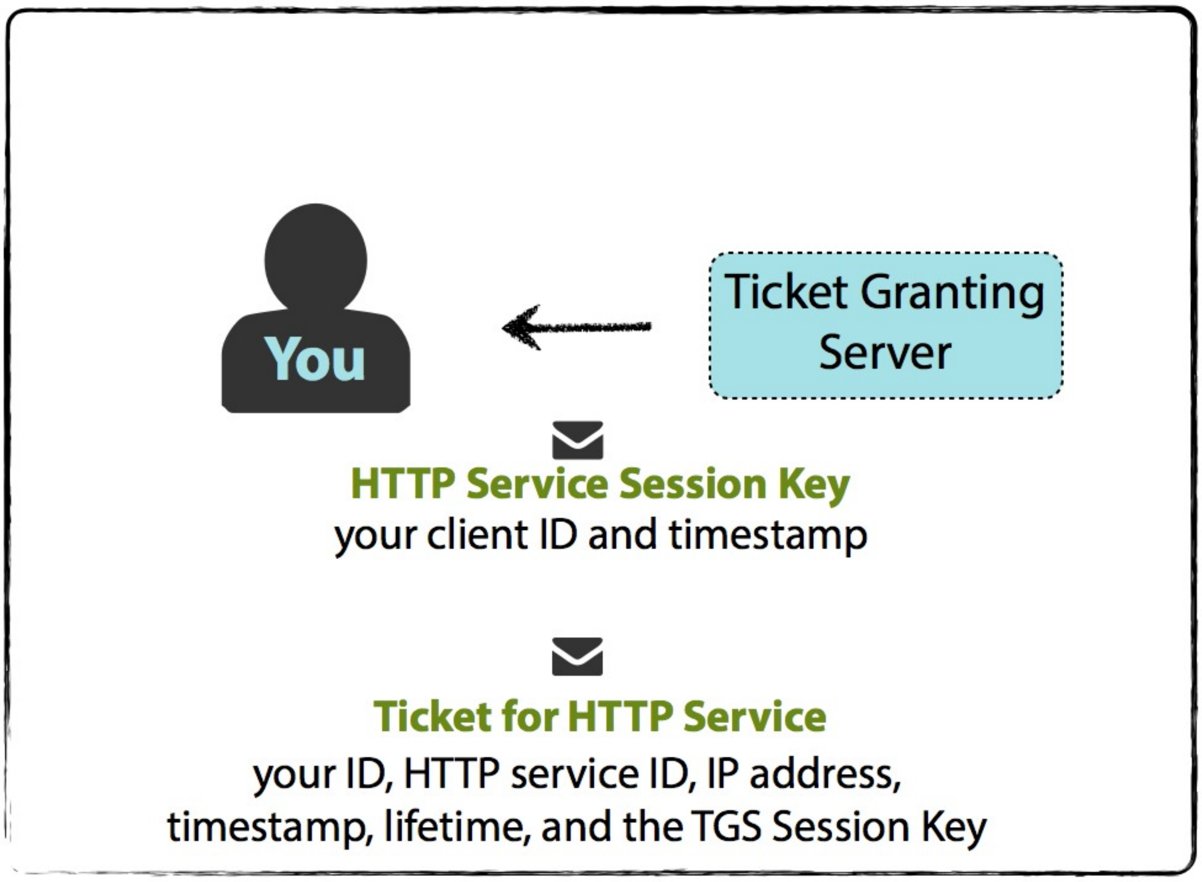
The TGS will then do the following:

- compare your client ID from the Authenticator to that of the TGT
- compare the timestamp from the Authenticator to that of the TGT (typical Kerberos-system tolerance of difference is 2 minutes, but can be configured otherwise)
- check to see if the TGT is expired (the lifetime element),
- check that the Authenticator is not already in the TGS's cache (for avoiding replay attacks), and
- if the network address in the original request is not null, compares the source's IP address to your network address (or within the requested list) within the TGT.

The Ticket Granting Server then randomly generates the HTTP Service Session Key, and prepares the HTTP Service ticket for you that contains:

- your name/ID,
- HTTP Service name/ID,
- your network address (may be a list of IP addresses for multiple machines, or may be null if wanting to use on any machine),
- timestamp,
- lifetime of the validity of the ticket, and
- HTTP Service Session Key,

and encrypts it with the HTTP Service Secret Key.


You
Ticket Granting Server
HTTP Service Session Key
your client ID and timestamp
Ticket for HTTP Service
your ID, HTTP service ID, IP address, timestamp, lifetime, and the TGS Session Key

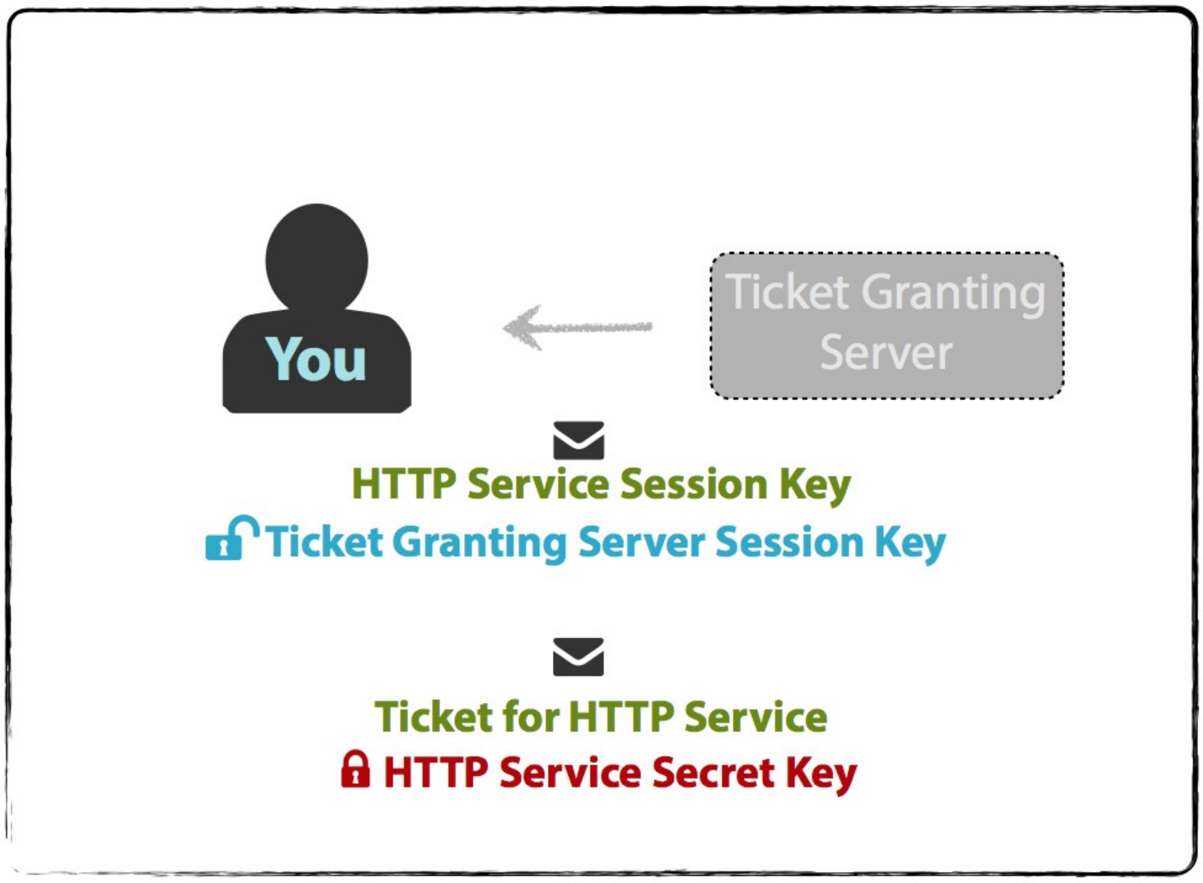
Then the TGS sends you two messages. One is the encrypted HTTP Service Ticket; the other contains:

- HTTP Service name/ID,
- timestamp,
- lifetime of the validity of the ticket, and
- HTTP Service Session Key,

that is encrypted with the TGS Session Key.

Your machine decrypts the latter message with the TGS Session Key that it cached earlier to obtain the HTTP Service Session Key.

Your machine can not, however, decrypt the HTTP Service Ticket since it’s encrypted with the HTTP Service Secret Key.

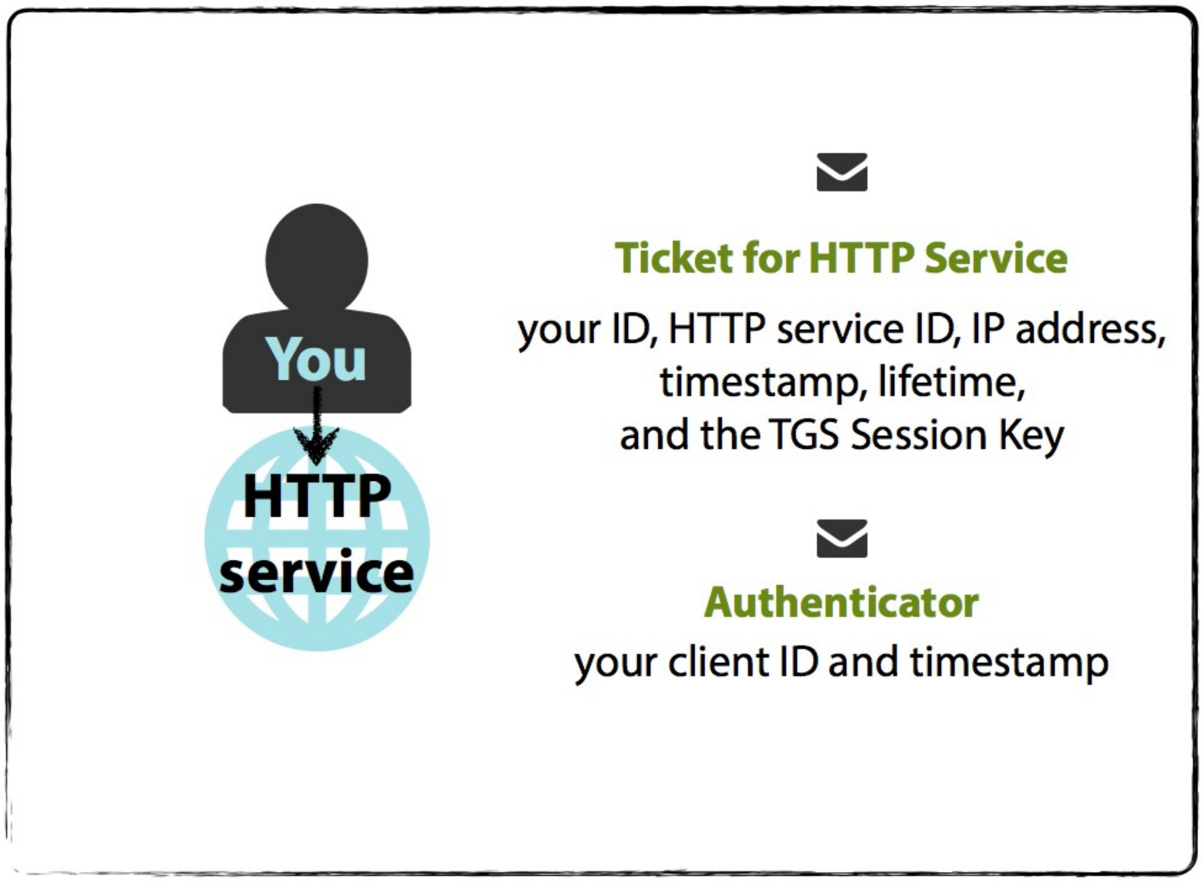


You and the HTTP Service

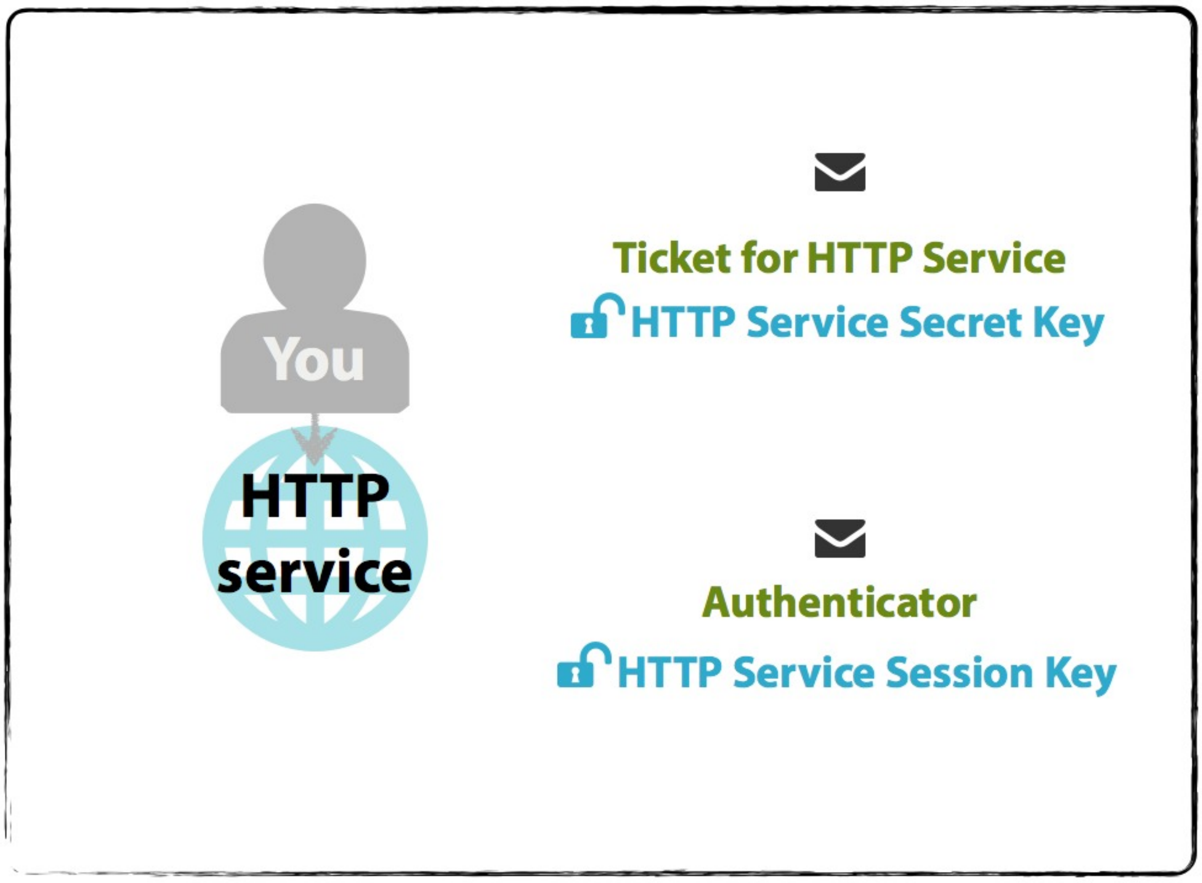
To now access the HTTP Service, your machine prepares another Authenticator message that contains:

- your name/ID,
- timestamp,

and is encrypted with the HTTP Service Session Key. Your machine then sends the Authenticator and the still-encrypted HTTP Service Ticket received from the TGS.


You
HTTP service
Ticket for HTTP Service
your ID, HTTP service ID, IP address, timestamp, lifetime, and the TGS Session Key
Authenticator
your client ID and timestamp

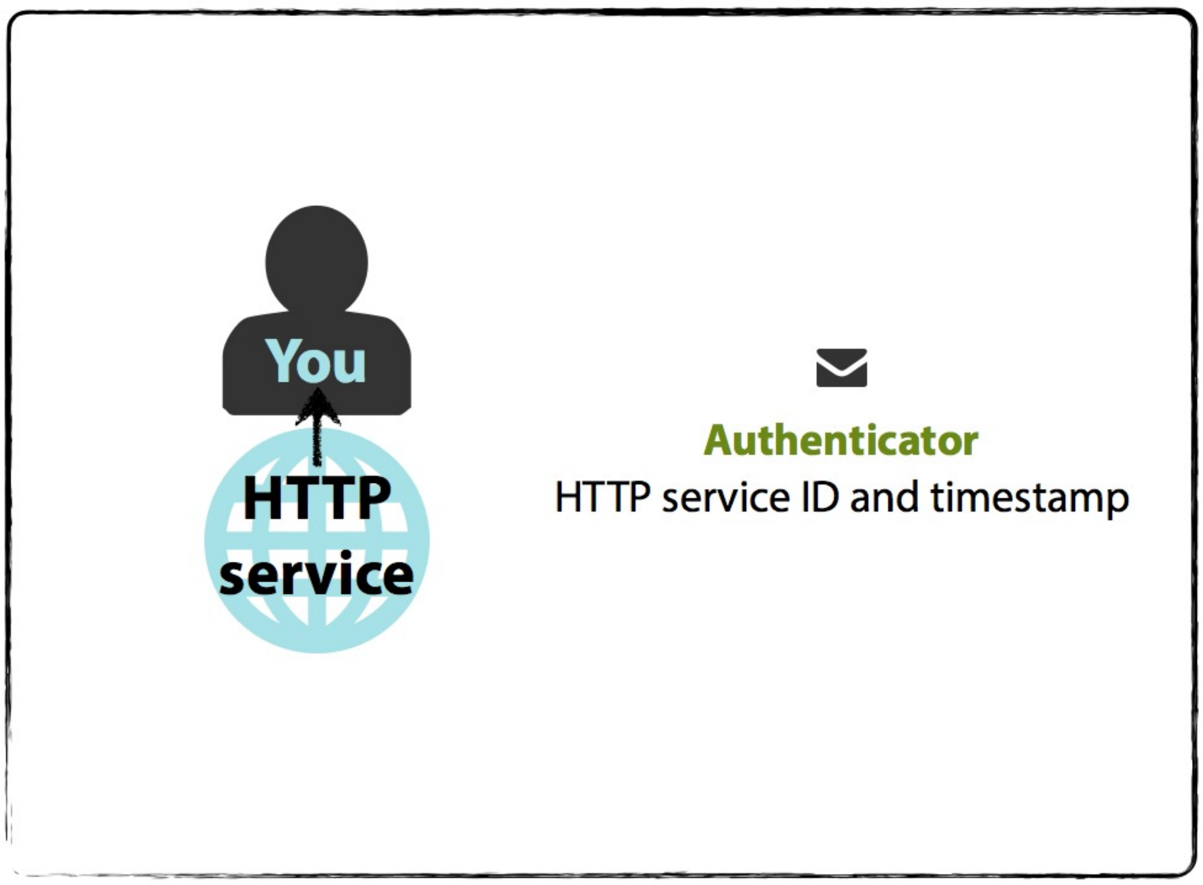
The HTTP Service then decrypts the Ticket with its Secret Key to obtain the HTTP Service Session Key. It then uses that Session Key to decrypt the Authenticator message you sent.



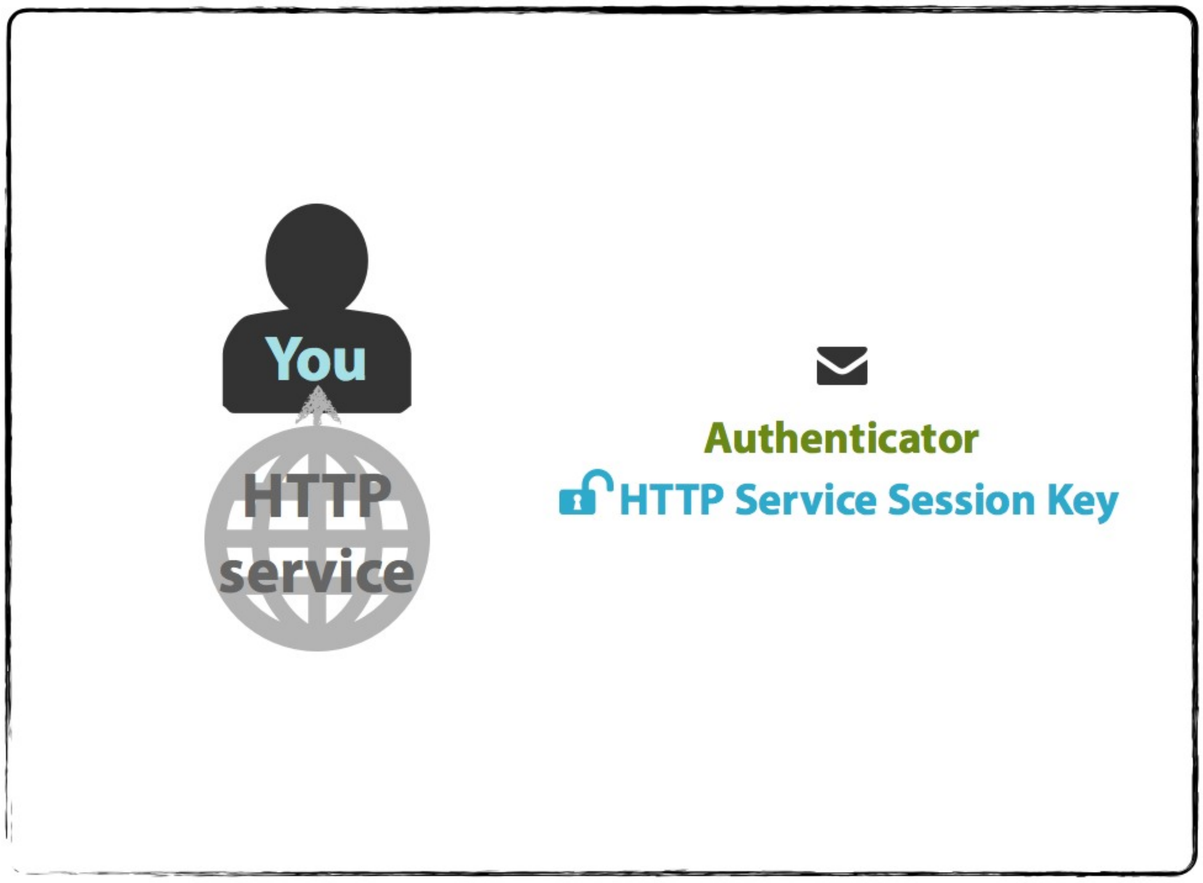
Similar to the TGS, the HTTP Server will then do the following:

- compares your client ID from the Authenticator to that of the Ticket,
- compares the timestamp from the Authenticator to that of the Ticket (typical Kerberos-system tolerance of difference is 2 minutes, but can be configured otherwise),
- checks to see if the Ticket is expired (the lifetime element),
- checks that the Authenticator is not already in the HTTP Server's cache (for avoiding replay attacks), and
- if the network address in the original request is not null, compares the source's IP address to your network address (or within the requested list) within the Ticket.

The HTTP Service then sends an Authenticator message containing its ID and timestamp in order to confirm its identity to you and is encrypted with the HTTP Service Session Key.

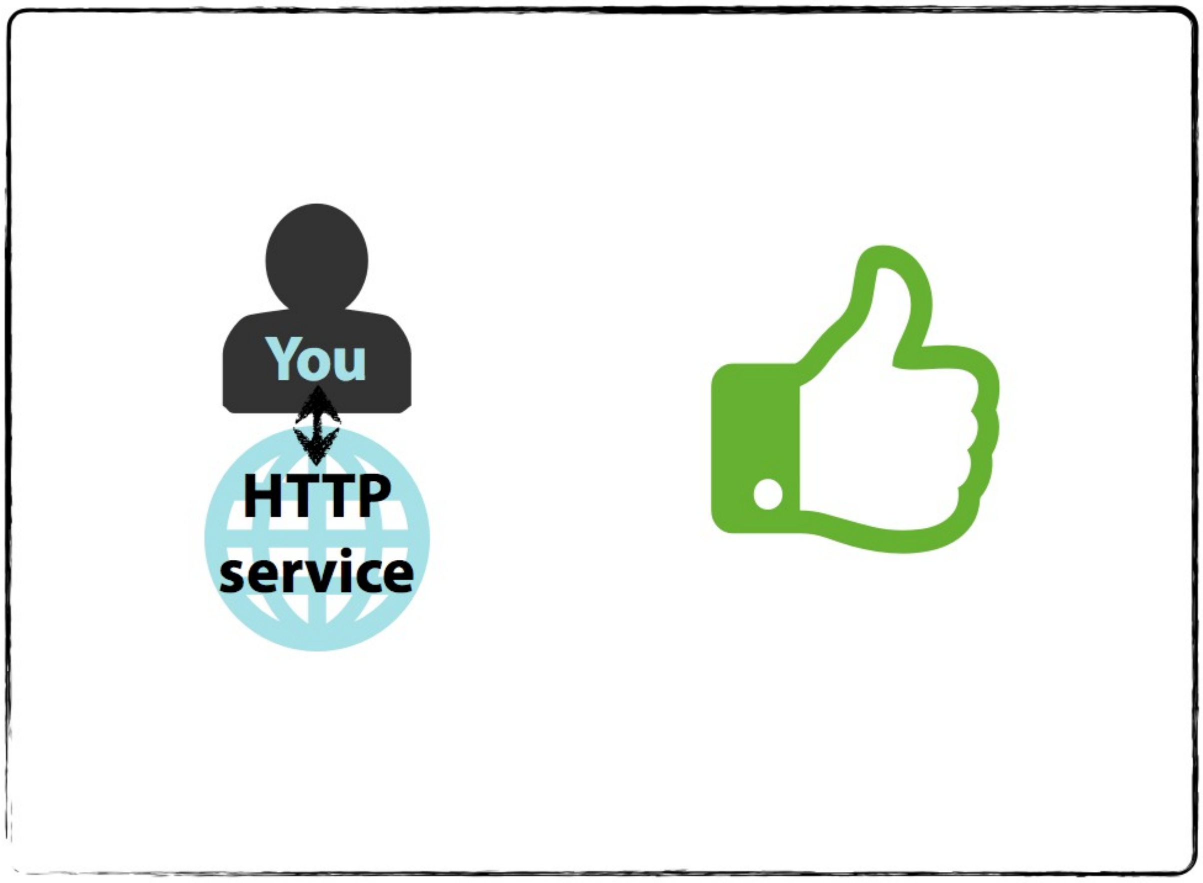


Your machine reads the Authenticator message by decrypting with the cached HTTP Service Session Key, and knows that it has to receive a message with the HTTP Service's ID and timestamp.



And now you have been authenticated to use the HTTP Service. Future requests use the cached HTTP Service Ticket, so long as it has not expired as defined within the lifetime attribute.

While I will write on this later, the HTTP Service itself must be able to support Kerberos. As well, you must also have a browser that supports SPNEGO/Negotiate.



Perhaps re-read the points previously outlined; check out this or this current implementation, especially the one on which I am paid to work that communicates with this popular implementation; or review a tutorial, resource guide, the go-to video that was sent to me when I started learning about Kerberos, or the RFC itself.

The above images were rendered with Keynote with icons used from font awesome and glyphicons, and are available on slideshare.

~

In a future posts, I'll write up how to actually setup a Kerberos realm, setting up an HTTP Service that will accept Negotiate authentication, and writing a web application that can plug into Kerberos for its authentication.

Update: May 16th, 2013: My post on setting up a web application + Apache for Kerberos, along with creating your own Kerberos test environment.

---

kerberoseli5authentication

∴

Has this article been helpful for you? Consider expressing your gratitude!

Need some help? I'm available for tutoring, mentoring, and interview prep!