

The Moron's Guide to Kerberos, Version 1.2.2

(That's Version 1.2.2 of the Guide, not Kerberos)

What follows is a brief guide to Kerberos: what it's for, how it works, how to use it. It is not for system administrators who want to know why they can't make the latest release, nor is it for applications programmers who want to know how to use the interface. It certainly isn't for Kerberos hackers. You know who you are.

[Brian Tung](#)
brian@isi.edu

Note: I've recently added a short tutorial on [ASN.1](#), since this important addition to Version 5 seems to be underdocumented.

Table of Contents

[What is Kerberos?](#)

[What does Kerberos do?](#)

[Assumptions Kerberos Makes](#)

[The \(High Level\) Details](#)

[Service Authentication](#)

[The Ticket Granting Server](#)

[Cross Realm Authentication](#)

[How to Use Kerberos](#)

[For The User](#)

[For The System Administrator](#)

[More Advanced Topics](#)

[Ticket Flags, and What They Mean](#)

Initial Tickets

Pre-Authenticated Tickets

Invalid Tickets

(Potentially) Postdated Tickets

Renewable Tickets

Proxy Tickets

Forwarded Tickets

What is Kerberos?



Kerberos is an authentication service developed at MIT. Its purpose is to allow users and services to *authenticate* themselves to each other. That is, it allows them to demonstrate their identity to each other, unequivocally (it is hoped).

MIT distributes Kerberos for free, in the hopes that it will replace the current industry ``standard," which is known, somewhat tongue-in-cheek, as *authentication by assertion*. Authentication by assertion works as follows. When a user runs a program than accesses a network service, the program (called the *client*) asserts to the service that it is running on behalf of the

user. That's it.

Needless to say, this provides a very low level of security. Consider, for example, the example of Berkeley `rlogin`. If a user `rlogins` to an account under his own name, but on another machine, and if the user's `.rhosts` has been set correctly, the `rlogin` program will assert the user's identity to the `rlogin` daemon on the remote machine, and that daemon will not require a password at all for login! This is disastrous if an attacker is somehow able either to convince the `rlogin` program that he is the legitimate user, or to rewrite a mutant version of `rlogin` that will assert that identity to the remote machine.

The alternative, to require entry of the user's password for each access to a network service, has at least two shortcomings. First of all, it is time-consuming for the user. Secondly, and more importantly, it is insecure when accessing services on a remote machine. For instance, if you are already logged into a remote machine, and decide to login from there to another remote machine, then your password would travel to the first remote machine ``in the clear" (unencrypted). Clearly, this is unacceptable.

Hence, Kerberos was designed to eliminate the need to demonstrate possession of private or secret information (the password) by divulging the information itself. Kerberos is based on the key distribution model developed by Needham and Schroeder [1]. A *key* is used to encrypt and decrypt short messages, and is itself typically a short sequence of bytes. Keys provide the basis for the authentication in Kerberos.

Roughly speaking, an encryption routine takes an encryption key and a *plaintext* message, and returns *ciphertext*. This ciphertext typically looks like garbage (random stream of bytes). Conversely, the decryption routine takes a decryption key and the ciphertext, and returns (if decryption is successful) the original plaintext. In Kerberos, at the present time, the encryption key and the decryption key are identical.

This is the hallmark of conventional cryptography, in which the keys are either identical or at least easily derivable from one another. In contrast, in public key cryptography, there are two keys, one for encryption, one for decryption, that are not derivable from one another. In fact, in many realizations, either key can be used for encryption, and the other key for decryption.

Kerberos is now in Version 5, Release 1! You can find out all about it, including how to get it, at its [MIT web site](http://www.mit.edu/~kerberos/).

What does Kerberos do?

How does Kerberos allow you to authenticate yourself? Well, how do you authenticate yourself in real life? Typically, you show your driver's license--or ID card, if you're not of driving age. (By the way, it's been a long time since I was either 5'8", or 110 pounds. Just in case you were wondering.)

What does this show? It shows that there is an agency (the one that issued the license or card) that has linked a given identity to a physical likeness. This physical likeness usually consists of a photo and some physical stats, and is considered to be uncopyable. (That is, you can't change yourself to look like someone else.)



The identity consists of a name and an address, and some other information, such as a birthdate. In addition, there may be some restrictions on what the named person can do; for instance, he or she may be required to wear corrective lenses while driving. (In many cases, this restriction is implicit: one can't drink until the age of 21, based on the birthdate on the card.) Finally, the identification has a limited lifetime, represented by the expiration date on the card.

Note that this demonstration of identity is contingent on a number of things. First of all, the card must not be tampered with (such as changing the birthdate, or the name, or the photo). Secondly, the person performing the authentication must accept the agency that issued the card. (Many supermarkets won't accept out-of-state or even out-of-town ID's when verifying checks.) There are some other concerns, such as that the person being authenticated really *hasn't* changed in appearance or name, or that the card is stolen, and so forth.

Kerberos works in basically the same way. It's typically used when a user on a network is attempting to make use of a network service, and the service wants assurance that the user is who he says he is. To that end, the user presents a *ticket* that is issued by the Kerberos *authentication server* (AS), much as a driver's license is issued by the DMV. The service then examines the ticket to verify the identity of the user. If all checks out, then the user is accepted.

Therefore, this ticket must contain information linking it unequivocally to the user. Since the user and the service don't meet face to face (whatever that would mean), a photo is of no use. No, the ticket must demonstrate that the bearer knows something only its intended user would know, such as a password. Furthermore, there must be safeguards against an attacker stealing the ticket, and using it later.

Assumptions Kerberos Makes

Kerberos does make some assumptions about the environment it lives in. It assumes that users won't make poor choices for passwords. If a user selects a password like "password" or "nothing," then an attacker who intercepts a few encrypted messages will be able to mount a *dictionary attack*, in which he tries password after password to see if it decrypts messages correctly. Success means that the user's password has been guessed and that the attacker can now impersonate the user to any verifier.

Similarly, Kerberos assumes that the workstations or machines are more or less secure, that only the network connections are vulnerable to compromise. In other words, Kerberos assumes that there is no way for an attacker to position himself between the user and the client in order to obtain the password in that manner.

The (High Level) Details

Here's how Kerberos goes about things. Both the user and the service are required to have keys registered with the AS. The user's key is derived from a password that he chooses; the service key is a randomly selected key (since no person is available to type in a password).

For the purposes of this explanation, let us imagine that messages are written on paper (instead of being electronic), and are "encrypted" by being locked in a strongbox by means of a key. In this "box world," principals are initialized by making a physical key and registering a copy of the key with the AS.

1. First the user sends a message to the AS: "I, J Random User, would like to talk to Foo Server."
2. When the AS receives this message, it makes up two copies of a brand new key. This is called the *session key*. It will be used in the direct exchange between user and service.

3. It puts one of the session keys in Box 1, along with a piece of paper with the name ``Foo Server'' written on it. It locks this box with the user's key.

Why is this piece of paper here? Recall that this box is really just an encrypted message, and that the session key is really just a sequence of random bytes. If Box 1 only contained the session key, then the user wouldn't be able to tell whether the response came back from the AS, or whether the decryption was successful. By putting in ``Foo Server,'' the user (or more precisely, the user's program) will be able to verify both that the box comes from the AS, and that the decryption was successful.

4. It puts the other session key in a Box 2, along with a piece of paper with the name ``J Random User'' written on it. It locks this box with the service's key.
5. It returns both boxes to the user.

In version 4, Box 2 was placed (unnecessarily) in Box 1.

6. The user unlocks Box 1 with his key, extracting the session key and the paper with ``Foo Server'' written on it.
7. The user can't open Box 2 (since it's locked with the service's key). Instead, he puts a piece of paper with the current time written on it in Box 3, and locks it with the session key. He then hands both boxes to the service.
8. The service opens the Box 2 with its own key, extracting the session key and the paper with ``J Random User'' written on it. It then opens Box 3 with the session key to extract the piece of paper with the current time on it. These items demonstrate the identity of the user.

The timestamp is put in Box 3 to prevent someone else from copying Box 2 (remember, these are simply electronic messages) and using it to impersonate the user at a later time. Because clocks don't always work in perfect synchrony, a small amount of leeway (about five minutes is typical) is given between the timestamp and the current time. In addition, the service maintains a list of recently sent authenticators, to make sure that they aren't resent in quick order.

You may wonder how the service is able to open Box 2, if there isn't anyone ``back there'' to type in a password. Well, the service key isn't derived from a password. Instead, it's randomly generated, then stored in a special file called a service key file. This file is assumed to be secure, so that no one can copy the file and impersonate the service to a legitimate user.

In Kerberos parlance, Box 2 is called the *ticket*, and Box 3 is called the *authenticator*. The authenticator typically contains more information than what is listed here. Some of this added information arises from the fact that this is an electronic message (for example, there is a checksum). There may also be an encryption key in the authenticator to provide for privacy in future communications between the user and the service.

There is a version of Kerberos called Bones, which is exactly like Kerberos, except that Bones doesn't lock any of the Boxes (1 or 2 for the AS, and 3 for the client). So what is it good for? The United States restricts export of cryptography; it's treated as munitions, in fact. There are mechanisms to get cryptographic software out, but there are an extraordinary number of hoops you have to leap through in order to do this. On the other hand, there is a wide variety of legitimate software that is exported, and expects Kerberos

to be there. Export versions of this software are shipped with Bones installed, ``tricking" them into thinking Kerberos is there. (I haven't seen a logo for Bones, but I expect someone could come up with something fairly exciting based on the Kerberos logo.)

Doug Rickard (dricard@technet2000.com.au) wrote to explain how Bones got its name. In 1988, he was working at MIT, with the Athena group. He was trying to get permission from the State Department to export Kerberos to Bond University in Australia. The State Department wouldn't allow it--not with DES included. To get it out of the country, they had to not only remove all calls to DES routines, but all comments and textual references to them as well, so that (superficially, at least) it was non-trivial to determine where the calls were originally placed.

To strip out all the DES calls and garbage, John Kohl wrote a program called `piranha`. At one of their progress meetings, Doug jokingly said, ``And we are left with nothing but the Bones." For lack of a better term, he then used the word ``Bones" and ``boned" in the meeting minutes to distinguish between the DES and non-DES versions of Kerberos. ``It somehow stuck," he says, ``and I have been ashamed of it ever since."

Back at Bond University, Errol Young then put encryption back into Bones, thus creating Encrypted Bones, or E-Bones.

Service Authentication

Sometimes, the user may want the service to be authenticated in return. To do so, the service takes the timestamp from the authenticator (Box 3), places it in Box 4, along with a piece of paper with ``Foo Server" written on it, locks it with the session key, and returns it to the user. (Clearly, it must include *something* with the timestamp; otherwise, it could simply return Box 3!)

It used to be the case (in version 4 of Kerberos) that the service would instead add 1 to the timestamp value, and return it, encrypted in the session key. That has been changed in version 5 of Kerberos.

The Ticket Granting Server

There is a subtle problem with the above exchange. It is used everytime a user wants to contact a service. But notice that he then has to enter in a password (unlock Box 1 with the key) each time. The obvious way around this is to cache the key derived from the password. But caching the key is dangerous. With a copy of this key, an attacker could impersonate the user at any time (until the password is next changed).

Kerberos resolves this problem by introducing a new agent, called the *ticket granting server* (TGS). The TGS is logically distinct from the AS, although they may reside on the same physical machine. (They are often referred to collectively as the KDC--the Key Distribution Center, from Needham and Schroeder [1].) The function of the TGS is as follows. Before accessing any regular service, the user requests a ticket to contact the TGS, just as if it were any other service. This ticket is called the *ticket granting ticket* (TGT).

After receiving the TGT, any time that the user wishes to contact a service, he requests a ticket not from the AS, but from the TGS. Furthermore, the reply is encrypted not with the user's secret key, but with the session key that the AS provided for use with the TGS. Inside that reply is the new session key for

use with the regular service. The rest of the exchange now continues as described above.

It's sort of like when you visit some workplaces. You show your regular ID to get a guest ID for the workplace. Now, when you want to enter various rooms in the workplace, instead of showing your regular ID over and over again, which might make it vulnerable to being dropped or stolen, you show your guest ID, which is only valid for a short time anyway. If it were stolen, you could get it invalidated and be issued a new one quickly and easily, something that you couldn't do with your regular ID.

Of course, there is a difference. In this analogy, an agency such as the DMV issues you your regular ID, and the workplace issues the guest ID. These are logically and physically distinct entities. On the other hand, in the TGT exchange, the AS and the TGS are logically distinct but are usually physically identical (same process).

The advantage this provides is that while passwords usually remain valid for months at a time, the TGT is good only for a fairly short period, typically eight hours. Afterwards, the TGT is not usable by anyone, including the user or any attacker. This TGT, as well as any tickets that you obtain using it, are stored in the *credentials cache*. There are a number of commands that you can use to manipulate your own credentials cache, which we'll get to in a moment.

The term ``credentials" actually refers to both the ticket and the session key in conjunction. However, you will often see the terms ``ticket cache" and ``credentials cache" used more or less interchangeably.

Cross Realm Authentication

So far, we've considered the case where there is a single AS and a single TGS, which may or may not reside on the same machine. As long as the number of requests is small, this is not a problem. But as the network grows, the number of requests grows with it, and the AS/TGS becomes a bottleneck in the authentication process. In other words, this system doesn't *scale*, which is bad for a distributed system such as Kerberos.

Therefore, it is often advantageous to divide the network into *realms*. These divisions are often made on organizational boundaries, although they need not be. Each realm has its own AS and its own TGS.

To allow for cross-realm authentication--that is, to allow users in one realm to access services in another--it is necessary first for the user's realm to register a *remote* TGS (RTGS) in the service's realm.

Recall that when the TGS was added, an additional exchange was added to the protocol. Here, yet another exchange is added: First, the user contacts the AS to access the TGS. Then the user contacts the TGS to access the RTGS. Finally, the user contacts the RTGS to access the actual service.

Actually, it can be worse than that. In some cases, where there are many realms, it is inefficient to register each realm in every other realm. Instead, there is a hierarchy of realms, so that in order to contact a service in another realm, it may be necessary to contact the RTGS in one or more intermediate realms. The names of each of these realms is recorded in the ticket.

This feature is new to Kerberos in version 5. In version 4, only peer-to-peer cross-realm authentication was permitted. In other words, in an environment with 100 realms, complete authentication coverage required the registration of $100 \cdot 99 = 9900$ remote TGS. This is called not scaling.

How to Use Kerberos

Kerberos does no one any good if they don't use it. However, it's easy to use--for the user, at least.

For The User

For you to use Kerberos, you must first establish a Kerberos *principal*. A Kerberos principal is something like a regular account on a machine. The name of the principal usually looks something like `your_name@YOUR.REALM`. The part before the at-sign is a string that you choose. Usually, it's the same as your regular account name. The part after the at-sign is the name of the realm. It may look like the name of the machine that your account is on, but there's no other resemblance there.

Associated with each principal is the name, a password, and some other information that we won't be concerned with right now. This information is stored in the Kerberos database, and is encrypted using a Kerberos master key. Therefore, it can't be examined by just anyone.

For the user, Kerberos is nearly transparent. There are a number of services set up which require Kerberos authentication. An example is `rlogin`. To use one of these services, you need to obtain a TGT first. The command for this is `kinit`:

```
% kinit
Password for your_name@YOUR.REALM:
```

When you enter in your password, the `kinit` program submits a request to the AS for a TGT. The password is used to compute a key, which in turn is used to decrypt part of the response from the AS. (This is the part that contains the confirmation of the request, as well as the session key.) If your password is entered in correctly, you now have a TGT. You can verify this by using the command `klist`:

```
% klist
Ticket cache: /var/tmp/krb5cc_1234
Default principal: your_name@YOUR.REALM

Valid starting    Expires          Service principal
24-Jul-95 12:58:02 24-Jul-95 20:58:15  krbtgt/YOUR.REALM@YOUR.REALM
```

The ticket cache field tells you which file contains your credentials cache. The default principal is the principal that the TGT is for (you). The remainder of the output is a list of your existing tickets. Since you've just started, there's only one. The service principal (`krbtgt`, etc) shows that this ticket is a TGT. Note that it's good for a short time: in this case, eight hours (more or less).

If you now use the Kerberos version of `rlogin`, this program will use the TGT in your credentials cache to request a ticket for the `rlogin` daemon on the machine you're logging into. This happens automatically, so all you see is the following:

```
% rlogin newhost.domain
Last login: Fri Jul 21 12:04:40 from etc etc
```

The only way you'll notice something's different is to get a listing of your cache:

```
% klist
```

```
Ticket cache: /var/tmp/krb5cc_1234
Default principal: your_name@YOUR.REALM
```

```
Valid starting      Expires            Service principal
24-Jul-95 12:58:02  24-Jul-95 20:58:15  krbtgt/YOUR.REALM@YOUR.REALM
24-Jul-95 13:03:33  24-Jul-95 20:58:15  host/newhost.domain@YOUR.REALM
```

The significance of the service principal is as follows. The first component (the part before the slash) is the base principal name. The second component (between the slash and the at-sign) is called the instance. For services, this is usually the hostname that the service is running on, although in the case of Kerberos services, it's the realm name. For users, it's usually null (in which case, there's no slash, either), or when the user is accessing some privileged item, some tag to indicate this (such as your_name/admin or your_name/secure). The last component (after the at-sign) is the realm name, as before.

The default action for `rlogin` is to leave any tickets that it obtains in the cache, so even though you strictly don't need it anymore, the ticket remains. This isn't a security problem unless someone can commandeer a terminal/station that you're currently logged into, which is nothing new.

Nevertheless, you may wish not to leave credentials lying around in your cache, in which case you can perform a `kdestroy`:

```
% kdestroy
% klist
klist: No credentials cache file found while setting cache flags
(ticket cache /var/tmp/krb5cc_1234)
```

`kdestroy` removes all tickets (including the TGT) from your cache.

For The System Administrator

For the system administrator, matters are more complex. The AS and the TGS (usually the same executable) must be configured and started. Principals must be registered. And most importantly, services must be made available that take advantage of Kerberos.

Starting the Kerberos server. Typically, the system administrator takes the following steps to start the Kerberos server:

1. Install the binaries. Clients should be placed in the common path. Certain binaries (such as `ksu` and all the root processes) must be owned by root, otherwise they will not operate correctly. In addition, `ksu` must have the sticky bit set. Typically, `make install` from the source tree will do all this for you.
2. Edit the `krb5.conf` file. This file contains configuration options for the clients, such as where all the KDCs are, which KDC is the "local" one, which machines fall under which realms, and so forth. (In revisions before beta 5, this functionality was provided by two files, `krb.conf` and `krb.realms`.)
3. Edit the `.k5login` file. Each line consists of a single principal name, which indicates right to `ksu` to root. This is just a simple access control list.
4. Edit the `/etc/services` file. Either copy this from an existing system, or (I believe) there is a file in the existing Kerberos source tree that indicates which ports to fill in.
5. Edit the `/etc/inetd.conf` file. Ideally, most of the services should be shut off. There are some

Kerberized versions of BSD commands that may be included. To get the daemon to respond to this new version of the file, you must find out which process `inetd` is, then send a `kill -HUP` signal to that process.

6. Edit the `/etc/rc.<hostname>` file. Add lines to ensure that the `krb5kdc` and `kadmin` daemons are started when the machine is booted.
7. Create the database. (Instructions coming soon.)
8. Add entries for people and servers. (Instructions coming soon.)
9. Start up the `krb5kdc` and `kadmin` servers in the background. They do this by default now.

Registering principals. Once you have the database started, you can run `kadmin` to add principals.

Kerberizing applications. This last is the hardest part of using Kerberos. Tailoring an application to use Kerberos is called *Kerberizing* the application. It entails

1. Finding out the user's identity.
2. Locating the user's credentials cache.
3. Checking to see if the ticket for that service is present.
4. If not, using the TGT and session key to send a request to obtain one.

This is often a non-trivial programming task. Fortunately, standard with the Kerberos release is a collection of pre-Kerberized applications. These include POP and the Berkeley R-commands (such as `rlogin`, `rsh`, and so forth).

More Advanced Topics

Ticket Flags, and What They Mean

Initial Tickets. Some services may choose to accept only those tickets which were directly issued by the AS, because they are issued in direct response to the correct entering of a password, rather than use of the credentials cache. The `INITIAL` flag is set to indicate this. Typically, this is used for direct communications with a service, but this flag is also set in a TGT issued directly by the AS. Any tickets issued by way of the TGT, however, have the `INITIAL` flag cleared.

Pre-Authenticated Tickets. On occasion, the AS may request clients to do more than send a message to demonstrate their identity (step 1 in the above list). To illustrate that some validation method was used before the issuance of the initial ticket, the `PRE-AUTHENT` flag is set. Furthermore, if a piece of hardware (such as a *smart card*) is used that could only be used by the valid user, then the `HW-AUTHENT` flag is set.

Unlike the `INITIAL` flag, these flags can be set in any ticket; if a ticket is not an initial ticket, then it inherits the values of these flags from the preceding TGT.

Invalid Tickets. Suppose that you want to run a batch job at night, and that this job requires tickets to access various services. You don't want to be present when this job starts, but you also don't want a valid ticket to be lying around all day beforehand.

In that case, you can request Kerberos to issue a *postdated* ticket. This ticket has the `INVALID` flag set, indicating that this ticket is not (yet) valid. No server is allowed to accept a ticket with the `INVALID` flag set. When the client sends a request to the TGS, this flag can be cleared without the user having to be present. This mechanism also allows the ticket to be permanently invalidated if it is stolen before its intended start time.

There are other reasons for invalid tickets; we won't get into them now.

(Potentially) Postdated Tickets. Not all tickets are postdated, of course. A TGT sent to the TGS may have the `MAY-POSTDATE` flag set to indicate that the TGS has the option of issuing a postdated ticket in response. This postdated ticket has the `POSTDATED` flag set.

Since the client can't directly manipulate the ticket, it must request the `MAY-POSTDATE` flag by using the `ALLOW-POSTDATE` option in the ticket request. Remember that this means not that the TGT is postdatable, but that the tickets you get *using* the TGT are postdatable. You can't request a postdated TGT by this means.

Renewable Tickets. Some applications run on for a long time, and may need to authenticate themselves several times over that period. Keeping a long-lived ticket exposes those credentials to theft. You could request a series of short-lived tickets, but this requires using your secret key over and over, which is even worse.

Instead, Kerberos allows clients to *renew* tickets. The `RENEWABLE` flag indicates that a ticket is eligible for renewing. Such a ticket has two expiration times: one that tells when the ticket is scheduled to run out, and another that tells how much longer the ticket can be renewed for.

Example. Suppose that the current time is 8:00, and you have a ticket that is scheduled to expire at 9:00. This ticket also has a final expiration time of 21:00 (say). Suppose further that tickets can only be renewed for eight hours at a time. (This policy is typically set in the Kerberos database.)

This ticket can be renewed so that its new expiration time is 9:00 + eight hours, or 17:00. At that time, it can only be renewed for another four hours, since that will take it to its final expiration time of 21:00, at which time a new ticket will have to be requested.

Proxy Tickets. A client may sometimes wish a service to act on its behalf; this will require the service to obtain tickets as if it were the client. This is the case when this first service has privileged access to another service. To allow this, a TGT with the `PROXIABLE` flag set can be given to the service, who can then forward it to the second service. To prevent wholesale use of this mechanism, only network addresses listed in the ticket are allowed to present a proxy ticket.

The `PROXIABLE` flag does not indicate that it is OK to issue a TGT, only regular tickets.

When the TGS receives a TGT with the `PROXIABLE` flag set, it can issue a ticket for the second service with the `PROXY` flag set. This second service can screen for tickets with this flag set, and require proxy agents to provide additional authentication.

A form of proxies, called *restricted proxies*, is the base mechanism for NetCheque, an electronic payment system.

Forwarded Tickets. Forwarded tickets are just like proxy tickets, except that they *can* authorize issuance of TGT's. The corresponding flags are called `FORWARDABLE` and `FORWARDED`.

Other Information About Kerberos

You can find out more information about Kerberos at <http://nii.isi.edu/info/kerberos/>. In

particular, an excellent introductory article can be found at
<http://nii.isi.edu/publications/kerberos-neuman-tso.html>.

[1] R. M. Needham and M. D. Schroeder, ``Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, Vol. 21 (12), pp. 993-99.

Last modified 19 December 1996.