Kerberos: An Authentication Service for Open Network Systems

Jennifer G. Steiner

Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
steiner@ATHENA.MIT.EDU

Clifford Neuman†

Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195
bcn@CS.WASHINGTON.EDU

Jeffrey I. Schiller

Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
jis@ATHENA.MIT.EDU

Introduction

This paper gives an overview of *Kerberos*, an authentication system designed by Miller and Neuman[1] for open network computing environments, and describes our experience using it at MIT's Project Athena.[2] In the first section of the paper, we explain why a new authentication model is needed for open networks, and what its requirements are. The second section lists the components of the *Kerberos* software and describes how they interact in providing the authentication service. In Section 3, we describe the *Kerberos* naming scheme.

Section 4 presents the building blocks of *Kerberos* authentication – the *ticket* and the *authenticator*. This leads to a discussion of the two authentication protocols: the initial authentication of a user to *Kerberos* (analogous to logging in), and the protocol for mutual authentication of a potential consumer and a potential producer of a network service.

Kerberos requires a database of information about its clients; Section 5 describes the database, its management, and the protocol for its modification. Section 6 describes the *Kerberos* interface to its users, applications programmers, and administrators. In Section 7, we describe how the Project Athena *Kerberos* fits into the rest of the Athena environment. We also describe the interaction of different *Kerberos* authentication domains, or *realms*; in our case, the relation between the Project Athena *Kerberos* and the *Kerberos* running at MIT's Laboratory for Computer Science.

In Section 8, we mention open issues and problems as yet unsolved. The last section gives

† Clifford Neuman was a member of the Project Athena staff during the design and initial implementation phase of *Kerberos*.

the current status of *Kerberos* at Project Athena. In the appendix, we describe in detail how *Kerberos* is applied to a network file service to authenticate users who wish to gain access to remote file systems.

Conventions. Throughout this paper we use terms that may be ambiguous, new to the reader, or used differently elsewhere. Below we state our use of those terms.

User, Client, Server. By *user*, we mean a human being who uses a program or service. A *client* also uses something, but is not necessarily a person; it can be a program. Often network applications consist of two parts; one program which runs on one machine and requests a remote service, and another program which runs on the remote machine and performs that service. We call those the *client* side and *server* side of the application, respectively. Often, a *client* will contact a *server* on behalf of a *user*.

Each entity that uses the *Kerberos* system, be it a user or a network server, is in one sense a client, since it uses the *Kerberos* service. So to distinguish *Kerberos* clients from clients of other services, we use the term *principal* to indicate such an entity. Note that a *Kerberos* principal can be either a user or a server. (We describe the naming of *Kerberos* principals in a later section.)

Service vs. Server. We use *service* as an abstract specification of some actions to be performed. A process which performs those actions is called a *server*. At a given time, there may be several *servers* (usually running on different machines) performing a given *service*. For example, at Athena there is one BSD UNIX *rlogin* server running on each of our timesharing machines.

Key, Private Key, Password. *Kerberos* uses private key encryption. Each *Kerberos* principal is assigned a large number, its private key, known only to that principal and *Kerberos*. In the case of a user, the private key is the result of a one-way function applied to the user's *password*. We use *key* as shorthand for *private key*.

Credentials. Unfortunately, this word has a special meaning for both the Sun Network File System and the *Kerberos* system. We explicitly state whether we mean NFS credentials or *Kerberos* credentials, otherwise the term is used in the normal English language sense.

Master and Slave. It is possible to run *Kerberos* authentication software on more than one machine. However, there is always only one definitive copy of the *Kerberos* database. The machine which houses this database is called the *master* machine, or just the *master*. Other machines may possess read-only copies of the *Kerberos* database, and these are called *slaves*.

1. Motivation

In a non-networked personal computing environment, resources and information can be protected by physically securing the personal computer. In a timesharing computing environment, the operating system protects users from one another and controls resources. In order to determine what each user is able to read or modify, it is necessary for the timesharing system to identify each user. This is accomplished when the user logs in.

In a network of users requiring services from many separate computers, there are three approaches one can take to access control: One can do nothing, relying on the machine to which the user is logged in to prevent unauthorized access; one can require the host to prove its identity, but trust the host's word as to who the user is; or one can require the user to prove her/his identity for each required service.

In a closed environment where all the machines are under strict control, one can use the first approach. When the organization controls all the hosts communicating over the network, this is a reasonable approach.

In a more open environment, one might selectively trust only those hosts under organizational control. In this case, each host must be required to prove its identity. The *rlogin* and *rsh* programs use this approach. In those protocols, authentication is done by checking the Internet address from which a connection has been established.

In the Athena environment, we must be able to honor requests from hosts that are not under organizational control. Users have complete control of their workstations: they can reboot them, bring them up standalone, or even boot off their own tapes. As such, the third approach must be taken; the user must prove her/his identity for each desired service. The server must also prove its identity. It is not sufficient to physically secure the host running a network server; someone elsewhere on the network may be masquerading as the given server.

Our environment places several

requirements on an identification mechanism. First, it must be secure. Circumventing it must be difficult enough that a potential attacker does not find the authentication mechanism to be the weak link. Someone watching the network should not be able to obtain the information necessary to impersonate another user. Second, it must be reliable. Access to many services will depend on the authentication service. If it is not reliable, the system of services as a whole will not be. Third, it should be transparent. Ideally, the user should not be aware of authentication taking place. Finally, it should be scalable. Many systems can communicate with Athena hosts. Not all of these will support our mechanism, but software should not break if they did.

Kerberos is the result of our work to satisfy the above requirements. When a user walks up to a workstation s/he ‘‘logs in’’. As far as the user can tell, this initial identification is sufficient to prove her/his identity to all the required network servers for the duration of the login session. The security of *Kerberos* relies on the security of several authentication servers, but not on the system from which users log in, nor on the security of the end servers that will be used. The authentication server provides a properly authenticated user with a way to prove her/his identity to servers scattered across the network.

Authentication is a fundamental building block for a secure networked environment. If, for example, a server knows for certain the identity of a client, it can decide whether to provide the service, whether the user should be given special privileges, who should receive the bill for the service, and so forth. In other words, authorization and accounting schemes can be built on top of the authentication that *Kerberos* provides, resulting in equivalent security to the lone personal computer or the timesharing system.

2. What is *Kerberos*?

Kerberos is a trusted third-party authentication service based on the model presented by Needham and Schroeder.[3] It is trusted in the sense that each of its clients believes *Kerberos'* judgement as to the identity of each of its other clients to be accurate. Timestamps (large numbers representing the current date and time) have been added to the original model to aid in the detection of *replay*. Replay occurs when a message is stolen off the network and resent later. For a more complete description of replay, and other issues of authentication, see Voydock and Kent.[4]

2.1. What Does It Do?

Kerberos keeps a database of its clients and their *private keys*. The private key is a large number known only to *Kerberos* and the client it belongs to. In the case that the client is a user, it is an encrypted password. Network services requiring authentication register with *Kerberos*, as do clients wishing to use those services. The private keys are negotiated at registration.

Because *Kerberos* knows these private keys, it can create messages which convince one client that another is really who it claims to be. *Kerberos* also generates temporary private keys, called *session keys*, which are given to two clients and no one else. A session key can be used to encrypt messages between two parties.

Kerberos provides three distinct levels of protection. The application programmer determines which is appropriate, according to the requirements of the application. For example, some applications require only that authenticity be established at the initiation of a network connection, and can assume that further messages from a given network address originate from the authenticated party. Our authenticated network file system uses this level of security.

Other applications require authentication of each message, but do not care whether the content of the message is disclosed or not. For these, *Kerberos* provides *safe messages*. Yet a higher level of security is provided by *private messages*, where each message is not only authenticated, but also encrypted. Private messages are used, for example, by the *Kerberos* server itself for sending passwords over the network.

2.2. Software Components

The Athena implementation comprises several modules (see Figure 1). The *Kerberos* applications library provides an interface for application clients and application servers. It contains, among others, routines for creating or reading authentication requests, and the routines for creating safe or private messages.

- *Kerberos* applications library
- encryption library
- database library
- database administration programs
- administration server
- authentication server
- db propagation software
- user programs
- applications

Figure 1. *Kerberos* Software Components.

Encryption in *Kerberos* is based on DES, the Data Encryption Standard.[5] The encryption library implements those routines. Several methods of encryption are provided, with tradeoffs between speed and security. An extension to the DES Cypher Block Chaining (CBC) mode, called the Propagating CBC mode, is also provided. In CBC, an error is propagated only through the current block of the cipher, whereas in PCBC, the error is propagated throughout the message. This renders the entire message useless if an error occurs, rather than just a portion of it. The encryption library is an independent module, and may be replaced with other DES implementations or a different encryption library.

Another replaceable module is the database management system. The current Athena implementation of the database library uses *ndbm*, although Ingres was originally used. Other database management libraries could be used as well.

The *Kerberos* database needs are straightforward; a record is held for each principal, containing the name, private key, and expiration date of the principal, along with some administrative information. (The expiration date is the date after which an entry is no longer valid. It is usually set to a few years into the future at registration.)

Other user information, such as real name, phone number, and so forth, is kept by another server, the *Hesiod* nameserver.[6] This way, sensitive information, namely passwords, can be handled by *Kerberos*, using fairly high security measures; while the non-sensitive information kept by *Hesiod* is dealt with differently; it can, for example, be sent unencrypted over the network.

The *Kerberos* servers use the database library, as do the tools for administering the database.

The *administration server* (or KDBM server) provides a read-write network interface to the database. The client side of the program may be run on any machine on the network. The server side, however, must run on the machine housing the *Kerberos* database in order to make changes to the database.

The *authentication server* (or *Kerberos* server), on the other hand, performs read-only operations on the *Kerberos* database, namely, the authentication of principals, and generation of session keys. Since this server does not modify the *Kerberos* database, it may run on a machine housing a read-only copy of the master *Kerberos* database.

Database propagation software manages replication of the *Kerberos* database. It is possible to have copies of the database on several different machines, with a copy of the authentication server running on each machine. Each of these *slave* machines receives an update of the *Kerberos* database from the *master* machine at given intervals.

Finally, there are end-user programs for logging in to *Kerberos*, changing a *Kerberos* password, and displaying or destroying *Kerberos* *tickets* (tickets are explained later on).

3. *Kerberos* Names

Part of authenticating an entity is naming it. The process of authentication is the verification that the client is the one named in a request. What does a name consist of? In *Kerberos*, both users and servers are named. As far as the authentication server is concerned, they are equivalent. A name consists of a primary name, an instance, and a realm, expressed as *name.instance@realm* (see Figure 2).

bcn
treese.root
jis@LCS.MIT.EDU
rlogin.priam@ATHENA.MIT.EDU

Figure 2. *Kerberos* Names.

The *primary name* is the name of the user or the service. The *instance* is used to distinguish among variations on the primary name. For users, an instance may entail special privileges, such as the ‘‘root’’ or ‘‘admin’’ instances. For services in the Athena environment, the instance is usually the name of the machine on which the server runs. For example, the *rlogin* service has different instances on different hosts:

rlogin.priam is the *rlogin* server on the host named priam. A *Kerberos* ticket is only good for a single named server. As such, a separate ticket is required to gain access to different instances of the same service. The *realm* is the name of an administrative entity that maintains authentication data. For example, different institutions may each have their own *Kerberos* machine, housing a different database. They have different *Kerberos* realms. (Realms are discussed further in section 8.2.)

4. How It Works

This section describes the *Kerberos* authentication protocols. The following abbreviations are used in the figures.

c	->	client
s	->	server
addr	->	client's network address
life	->	lifetime of ticket
tgs, TGS	->	ticket-granting server
Kerberos	->	authentication server
KDBM	->	administration server
K_x	->	x's private key
$K_{x,y}$	->	session key for x and y
$\{abc\}K_x$	->	abc encrypted in x's key
$T_{x,y}$	->	x's ticket to use y
A_x	->	authenticator for x
WS	->	workstation

As mentioned above, the *Kerberos* authentication model is based on the Needham and Schroeder key distribution protocol. When a user requests a service, her/his identity must be established. To do this, a ticket is presented to the server, along with proof that the ticket was originally issued to the user, not stolen. There are three phases to authentication through *Kerberos*. In the first phase, the user obtains credentials to be used to request access to other services. In the second phase, the user requests authentication for a specific service. In the final phase, the user presents those credentials to the end server.

4.1. Credentials

There are two types of credentials used in the *Kerberos* authentication model: *tickets* and *authenticators*. Both are based on private key encryption, but they are encrypted using different keys. A ticket is used to securely pass the identity of the person to whom the ticket was issued between the authentication server and the end server. A ticket also passes information that can be used to make sure that the person using the ticket is the same person to which it was issued. The authenticator contains the additional information which, when compared against that in the ticket proves that the client presenting the ticket is the same one to which the ticket was issued.

A ticket is good for a single server and a single client. It contains the name of the server, the name of the client, the Internet address of the client, a timestamp, a lifetime, and a random session key. This information is encrypted using the key of the server for which the ticket will be used. Once the ticket has been issued, it may be used multiple times by the named client to gain access to the named server, until the ticket expires. Note that because the ticket is encrypted in the key of the server, it is safe to allow the user to pass the ticket on to the server without having to worry about the user modifying the ticket (see Figure 3).

$$\{s, c, addr, timestamp, life, K_{s,c}\}K_s$$

Figure 3. A *Kerberos* Ticket.

Unlike the ticket, the authenticator can only be used once. A new one must be generated each time a client wants to use a service. This does not present a problem because the client is able to build the authenticator itself. An authenticator contains the name of the client, the workstation's IP address, and the current workstation time. The authenticator is encrypted in the session key that is part of the ticket (see Figure 4).

$$\{c, addr, timestamp\}K_{s,c}$$

Figure 4. A *Kerberos* Authenticator.

4.2. Getting the Initial Ticket

When the user walks up to a workstation, only one piece of information can prove her/his identity: the user's password. The initial exchange with the authentication server is designed to minimize the chance that the password will be compromised, while at the same time not allowing a user to properly authenticate her/himself without knowledge of that password. The process of logging in appears to the user to be the same as logging in to a timesharing system. Behind the scenes, though, it is quite different (see Figure 5).

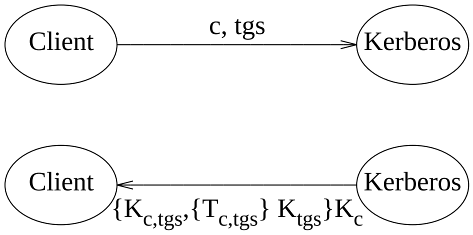


Figure 5. Getting the Initial Ticket.

The user is prompted for her/his username. Once it has been entered, a request is sent to the authentication server containing the user's name and the name of a special service known as the *ticket-granting service*.

The authentication server checks that it knows about the client. If so, it generates a random session key which will later be used between the client and the ticket-granting server. It then creates a ticket for the ticket-granting server which contains the client's name, the name of the ticket-granting server, the current time, a lifetime for the ticket, the client's IP address, and the random session key just created. This is all encrypted in a key known only to the ticket-granting server and the authentication server.

The authentication server then sends the ticket, along with a copy of the random session key and some additional information, back to the client. This response is encrypted in the client's private key, known only to *Kerberos* and the client, which is derived from the user's password.

Once the response has been received by the client, the user is asked for her/his password. The password is converted to a DES key and used to decrypt the response from the authentication server. The ticket and the session key, along with some of the other information, are stored for future use, and the user's password and DES key are erased from memory.

Once the exchange has been completed, the workstation possesses information that it can use to prove the identity of its user for the lifetime of the ticket-granting ticket. As long as the software on the workstation had not been previously tampered with, no information exists that will allow someone else to impersonate the user beyond the life of the ticket.

4.3. Requesting a Service

For the moment, let us pretend that the user already has a ticket for the desired server. In order to gain access to the server, the application builds an authenticator containing the client's name and IP address, and the current time. The authenticator is then encrypted in the session key that was received with the ticket for the server. The client then sends the authenticator along with the ticket to the server in a manner defined by the individual application.

Once the authenticator and ticket have been received by the server, the server decrypts the ticket, uses the session key included in the ticket to decrypt the authenticator, compares the information in the ticket with that in the authenticator, the IP address from which the request was received, and the present time. If everything matches, it allows the request to proceed (see Figure 6).

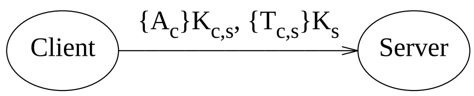


Figure 6. Requesting a Service.

It is assumed that clocks are synchronized to within several minutes. If the time in the request is too far in the future or the past, the server treats the request as an attempt to replay a previous request. The server is also allowed to keep track of all past requests with timestamps that are still valid. In order to further foil replay attacks, a request received with the same ticket and timestamp as one already received can be discarded.

Finally, if the client specifies that it wants the server to prove its identity too, the server adds one to the timestamp the client sent in the authenticator, encrypts the result in the session key, and sends the result back to the client (see Figure 7).

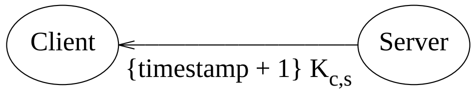


Figure 7. Mutual Authentication.

At the end of this exchange, the server is certain that, according to *Kerberos*, the client is who it says it is. If mutual authentication occurs, the client is also convinced that the server is authentic. Moreover, the client and server share a key which no one else knows, and can safely assume that a reasonably recent message encrypted in that key originated with the other party.

4.4. Getting Server Tickets

Recall that a ticket is only good for a single server. As such, it is necessary to obtain a separate ticket for each service the client wants to use. Tickets for individual servers can be obtained from the ticket-granting service. Since the ticket-granting service is itself a service, it makes use of the service access protocol described in the previous section.

When a program requires a ticket that has not already been requested, it sends a request to the ticket-granting server (see Figure 8). The request contains the name of the server for which a ticket is requested, along with the ticket-granting ticket and an authenticator built as described in the previous section.

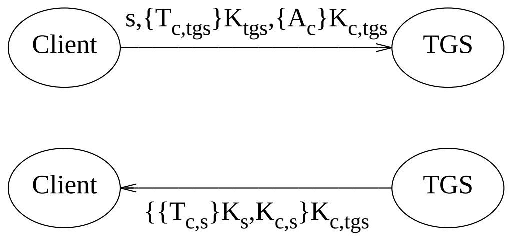
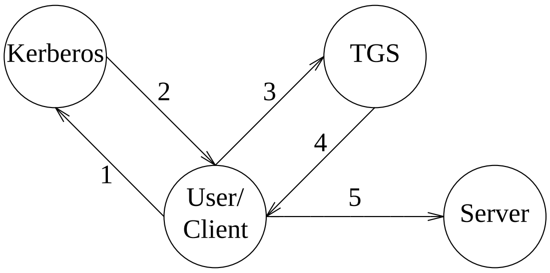


Figure 8. Getting a Server Ticket.

The ticket-granting server then checks the authenticator and ticket-granting ticket as described above. If valid, the ticket-granting server generates a new random session key to be used between the client and the new server. It then builds a ticket for the new server containing the client's name, the server name, the current time, the client's IP address and the new session key it just generated. The lifetime of the new ticket is the minimum of the remaining life for the ticket-granting ticket and the default for the service.

The ticket-granting server then sends the ticket, along with the session key and other information, back to the client. This time, however, the reply is encrypted in the session key that was part of the ticket-granting ticket. This way, there is no need for the user to enter her/his password again. Figure 9 summarizes the authentication protocols.



1. Request for TGS ticket
2. Ticket for TGS
3. Request for Server ticket
4. Ticket for Server
5. Request for service

Figure 9. Kerberos Authentication Protocols.

5. The *Kerberos* Database

Up to this point, we have discussed operations requiring read-only access to the *Kerberos* database. These operations are performed by the authentication service, which can run on both master and slave machines (see Figure 10).

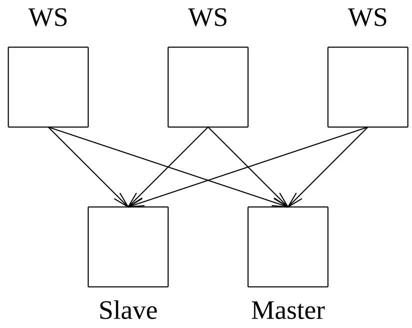


Figure 10. Authentication Requests.

In this section, we discuss operations that require write access to the database. These operations are performed by the administration service, called the *Kerberos* Database Management Service *(KDBM)*. The current implementation stipulates that changes may only be made to the master *Kerberos* database; slave copies are read-only. Therefore, the KDBM server may only run on the master *Kerberos* machine (see Figure 11).

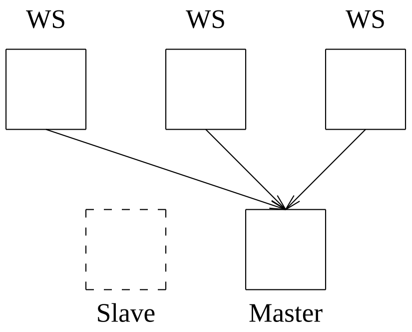


Figure 11. Administration Requests.

Note that, while authentication can still occur (on slaves), administration requests cannot be serviced if the master machine is down. In our experience, this has not presented a problem, as administration requests are infrequent.

The KDBM handles requests from users to change their passwords. The client side of this program, which sends requests to the KDBM over the network, is the *kpasswd* program. The KDBM also accepts requests from *Kerberos* administrators, who may add principals to the database, as well as change passwords for existing principals. The client side of the administration program, which also sends requests to the KDBM over the network, is the *kadmin* program.

5.1. The KDBM Server

The KDBM server accepts requests to add principals to the database or change the passwords for existing principals. This service is unique in that the ticket-granting service will not issue tickets for it. Instead, the authentication service itself must be used (the same service that is used to get a ticket-granting ticket). The purpose of this is to require the user to enter a password. If this were not so, then if a user left her/his workstation unattended, a passerby could walk up and change her/his password for them, something which should be prevented. Likewise, if an administrator left her/his workstation unguarded, a passerby could change any password in the system.

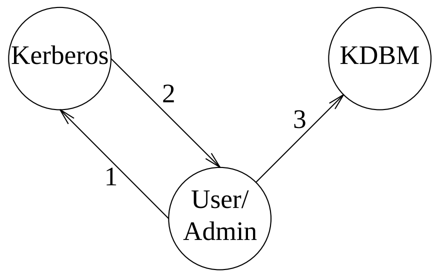
When the KDBM server receives a request, it authorizes it by comparing the authenticated principal name of the requester of the change to the principal name of the target of the request. If they are the same, the request is permitted. If they are not the same, the KDBM server consults an access control list (stored in a file on the master *Kerberos* system). If the requester's principal name is found in this file, the request is permitted, otherwise it is denied.

By convention, names with a **NULL** instance (the default instance) do not appear in the access control list file; instead, an **admin** instance is used. Therefore, for a user to become an administrator of *Kerberos* an **admin** instance for that username must be created, and added to the access control list. This convention allows an administrator to use a different password for *Kerberos* administration then s/he would use for normal login.

All requests to the KDBM program, whether permitted or denied, are logged.

5.2. The *kadmin* and *kpasswd* Programs

Administrators of *Kerberos* use the *kadmin* program to add principals to the database, or change the passwords of existing principals. An administrator is required to enter the password for their *admin* instance name when they invoke the *kadmin* program. This password is used to fetch a ticket for the KDBM server (see Figure 12).



1. Request for KDBM ticket
2. Ticket for KDBM
3. *kadmin* or *kpasswd* request

Figure 12. Kerberos Administration Protocol.

Users may change their *Kerberos* passwords using the *kpasswd* program. They are required to enter their old password when they invoke the program. This password is used to fetch a ticket for the KDBM server.

5.3. Database Replication

Each *Kerberos* realm has a *master Kerberos* machine, which houses the master copy of the authentication database. It is possible (although not necessary) to have additional, read-only copies of the database on *slave* machines elsewhere in the system. The advantages of having multiple copies of the database are those usually cited for replication: higher availability and better performance. If the master machine is down, authentication can still be achieved on one of the slave machines. The ability to perform authentication on any one of several machines reduces the probability of a bottleneck at the master machine.

Keeping multiple copies of the database introduces the problem of data consistency. We have found that very simple methods suffice for dealing with inconsistency. The master database is dumped every hour. The database is sent, in its entirety, to the slave machines, which then update their own databases. A program on the master host, called *kprop*, sends the update to a peer program, called *kpropd*, running on each of the slave machines (see Figure 13). First *kprop* sends a checksum of the new database it is about to send. The checksum is encrypted in the *Kerberos* master database key, which both the master and slave *Kerberos* machines possess. The data is then transferred over the network to the *kpropd* on the slave machine. The slave propagation server calculates a checksum of the data it has received, and if it matches the checksum sent by the master, the new information is used to update the slave's database.

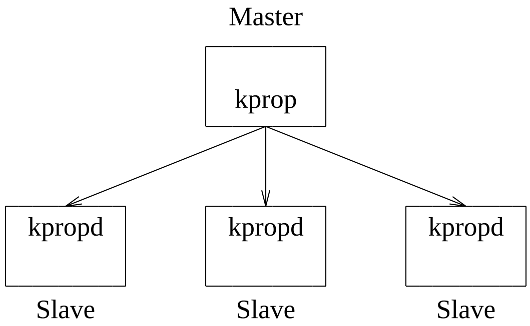


Figure 13. Database Propagation.

All passwords in the *Kerberos* database are encrypted in the master database key Therefore, the information passed from master to slave over the network is not useful to an eavesdropper. However, it is essential that only information from the master host be accepted by the slaves, and that tampering of data be detected, thus the checksum.

6. *Kerberos* From the Outside Looking In

The section will describe *Kerberos* from the practical point of view, first as seen by the user, then from the application programmer's viewpoint, and finally, through the tasks of the *Kerberos* administrator.

6.1. User's Eye View

If all goes well, the user will hardly notice that *Kerberos* is present. In our UNIX implementation, the ticket-granting ticket is obtained from *Kerberos* as part of the *login* process. The changing of a user's *Kerberos* password is part of the *passwd* program. And *Kerberos* tickets are automatically destroyed when a user logs out.

If the user's login session lasts longer than the lifetime of the ticket-granting ticket (currently 8 hours), the user will notice *Kerberos'* presence because the next time a *Kerberos*-authenticated application is executed, it will fail. The *Kerberos* ticket for it will have expired. At that point, the user can run the *kinit* program to obtain a new ticket for the ticket-granting server. As when logging in, a password must be provided in order to get it. A user executing the *klist* command out of curiosity may be surprised at all the tickets which have silently been obtained on her/his behalf for services which require *Kerberos* authentication.

6.2. From the Programmer's Viewpoint

A programmer writing a *Kerberos* application will often be adding authentication to an already existing network application consisting of a client and server side. We call this process ''Kerberizing'' a program. Kerberizing usually involves making a call to the *Kerberos* library in order to perform authentication at the initial request for service. It may also involve calls to the DES library to encrypt messages and data which are subsequently sent between application client and application server.

The most commonly used library functions are *krb_mk_req* on the client side, and *krb_rd_req* on the server side. The *krb_mk_req* routine takes as parameters the name, instance, and realm of the target server, which will be requested, and possibly a checksum of the data to be sent. The client then sends the message returned by the *krb_mk_req* call over the network to the server side of the application. When the server receives this message, it makes a call to the library routine *krb_rd_req*. The routine returns a judgement about the authenticity of the sender's alleged identity.

If the application requires that messages sent between client and server be secret, then library calls can be made to *krb_mk_priv (krb_rd_priv)* to encrypt (decrypt) messages in the session key which both sides now share.[7]

6.3. The *Kerberos* Administrator's Job

The *Kerberos* administrator's job begins with running a program to initialize the database. Another program must be run to register essential principals in the database, such as the *Kerberos* administrator's name with an **admin** instance. The *Kerberos* authentication server and the administration server must be started up. If there are slave databases, the administrator must arrange that the programs to propagate database updates from master to slaves be kicked off periodically.

After these initial steps have been taken, the administrator manipulates the database over the network, using the *kadmin* program. Through that program, new principals can be added, and passwords can be changed.

In particular, when a new *Kerberos* application is added to the system, the *Kerberos* administrator must take a few steps to get it working. The server must be registered in the database, and assigned a private key (usually this is an automatically generated random key). Then, some data (including the server's key) must be extracted from the database and installed in a file on the server's machine. The default file is */etc/srvtab*. The *krb_rd_req* library routine called by the server (see the previous section) uses the information in that file to decrypt messages sent encrypted in the server's private key. The */etc/srvtab* file authenticates the server as a password typed at a terminal authenticates the user.

The *Kerberos* administrator must also ensure that *Kerberos* machines are physically secure, and would also be wise to maintain backups of the Master database.[8]

7. The Bigger Picture

In this section, we describe how *Kerberos* fits into the Athena environment, including its use by other network services and applications, and how it interacts with remote *Kerberos* realms. For a more complete description of the Athena environment, please see G. W. Treese.[9]

7.1. Other Network Services' Use of *Kerberos*

Several network applications have been modified to use *Kerberos*. The *rlogin* and *rsh* commands first try to authenticate using *Kerberos*. A user with valid *Kerberos* tickets can rlogin to another Athena machine without having to set up *.rhosts* files. If the *Kerberos* authentication fails, the programs fall back on their usual methods of authorization, in this case, the *.rhosts* files.

We have modified the Post Office Protocol to use *Kerberos* for authenticating users who wish to retrieve their electronic mail from the ''post office''. A message delivery program, called *Zephyr*, has been recently developed at Athena, and it uses *Kerberos* for authentication as well.[10]

The program for signing up new users, called *register*, uses both the Service Management System (SMS)[11] and *Kerberos*. From SMS, it determines whether the information entered by the would-be new Athena user, such as name and MIT identification number, is valid. It then checks with *Kerberos* to see if the requested username is unique. If all goes well, a new entry is made to the *Kerberos* database, containing the username and password.

For a detailed discussion of the use of *Kerberos* to secure Sun's Network File System, please refer to the appendix.

7.2. Interaction with Other Kerberi

It is expected that different administrative organizations will want to use *Kerberos* for user authentication. It is also expected that in many cases, users in one organization will want to use services in another. *Kerberos* supports multiple administrative domains. The specification of names in *Kerberos* includes a field called the *realm*. This field contains the name of the administrative domain within which the user is to be authenticated.

Services are usually registered in a single realm and will only accept credentials issued by an authentication server for that realm. A user is usually registered in a single realm (the local realm), but it is possible for her/him to obtain credentials issued by another realm (the remote realm), on the strength of the authentication provided by the local realm. Credentials valid in a remote realm indicate the realm in which the user was originally authenticated. Services in the remote realm can choose whether to honor those credentials, depending on the degree of security required and the level of trust in the realm that initially authenticated the user.

In order to perform cross-realm authentication, it is necessary that the administrators of each pair of realms select a key to be shared between their realms. A user in the local realm can then request a ticket-granting ticket from the local authentication server for the ticket-granting server in the remote realm. When that ticket is used, the remote ticket-granting server recognizes that the request is not from its own realm, and it uses the previously exchanged key to decrypt the ticket-granting ticket. It then issues a ticket as it normally would, except that the realm field for the client contains the name of the realm in which the client was originally authenticated.

This approach could be extended to allow one to authenticate oneself through a series of realms until reaching the realm with the desired service. In order to do this, though, it would be necessary to record the entire path that was taken, and not just the name of the initial realm in which the user was authenticated. In such a situation, all that is known by the server is that A says that B says that C says that the user is so-and-so. This statement can only be trusted if everyone along the path is also trusted.

8. Issues and Open Problems

There are a number of issues and open problems associated with the *Kerberos* authentication mechanism. Among the issues are how to decide the correct lifetime for a ticket, how to allow proxies, and how to guarantee workstation integrity.

The ticket lifetime problem is a matter of choosing the proper tradeoff between security and convenience. If the life of a ticket is long, then if a ticket and its associated session key are stolen or misplaced, they can be used for a longer period of time. Such information can be stolen if a user forgets to log out of a public workstation. Alternatively, if a user has been authenticated on a system that allows multiple users, another user with access to root might be able to find the information needed to use stolen tickets. The problem with giving a ticket a short lifetime, however, is that when it expires, the user will have to obtain a new one which requires the user to enter the password again.

An open problem is the proxy problem. How can an authenticated user allow a server to acquire other network services on her/his behalf? An example where this would be important is the use of a service that will gain access to protected files directly from a fileserver. Another example of this problem is what we call *authentication forwarding*. If a user is logged into a workstation and logs in to a remote host, it would be nice if the user had access to the same services available locally, while running a program on the remote host. What makes this difficult is that the user might not trust the remote host, thus authentication forwarding is not desirable in all cases. We do not presently have a solution to this problem.

Another problem, and one that is important in the Athena environment, is how to guarantee the integrity of the software running on a workstation. This is not so much of a problem on private workstations since the user that will be using it has control over it. On public workstations, however, someone might have come along and modified the *login* program to save the user's password. The only solution presently available in our environment is to make it difficult for people to modify software running on the public workstations. A better solution would require that the user's key never leave a system that the user knows can be trusted. One way this could be done would be if the user possessed a *smartcard* capable of doing the encryptions required in the authentication protocol.

9. Status

A prototype version of *Kerberos* went into production in September of 1986. Since January of 1987, *Kerberos* has been Project Athena's sole means of authenticating its 5,000 users, 650 workstations, and 65 servers. In addition, *Kerberos* is now being used in place of *.rhosts* files for controlling access in several of Athena's timesharing systems.

10. Acknowledgements

Kerberos was initially designed by Steve Miller and Clifford Neuman with suggestions from Jeff Schiller and Jerry Saltzer. Since that time, numerous other people have been involved with the project. Among them are Jim Aspnes, Bob Baldwin, John Barba, Richard Basch, Jim Bloom, Bill Bryant, Mark Colan, Rob French, Dan Geer, John Kohl, John Kubiatowicz, Bob Mckie, Brian Murphy, John Ostlund Ken Raeburn, Chris Reed, Jon Rochlis, Mike Shanzer, Bill Sommerfeld, Ted T'so, Win Treese, and Stan Zanarotti.

We are grateful to Dan Geer, Kathy Lieben, Josh Lubarr, Ken Raeburn, Jerry Saltzer, Ed Steiner, Robbert van Renesse, and Win Treese whose suggestions much improved earlier drafts of this paper.

The illustration on the title page is by Betsy Bruemmer.

Appendix

Kerberos Application to SUN's Network File System (NFS)

A key component of the Project Athena workstation system is the interposing of the network between the user's workstation and her/his private file storage (home directory). All private storage resides on a set of computers (currently VAX 11/750s) that are dedicated to this purpose. This allows us to offer services on publicly available UNIX workstations. When a user logs in to one of these publicly available workstations, rather then validate her/his name and password against a locally resident password file, we use *Kerberos* to determine her/his authenticity. The *login* program prompts for a username (as on any UNIX system). This username is used to fetch a *Kerberos* ticket-granting ticket. The *login* program uses the password to generate a DES key for decrypting the ticket. If decryption is successful, the user's home directory is located by consulting the *Hesiod* naming service and mounted through NFS. The *login* program then turns control over to the user's shell, which then can run the traditional per-user customization files because the home directory is now ''attached'' to the workstation. The *Hesiod* service is also used to construct an entry in the local password file. (This is for the benefit of programs that look up information in */etc/passwd*.)

From several options for delivery of remote file service, we chose SUN's Network File System. However this system fails to mesh with our needs in a crucial way. NFS assumes that all workstations fall into two categories (as viewed from a file server's point of view): trusted and untrusted. Untrusted systems cannot access any files at all, trusted can. Trusted systems are completely trusted. It is assumed that a trusted system is managed by friendly management. Specifically, it is possible from a trusted workstation to masquerade as any valid user of the file service system and thus gain access to just about every file on the system. (Only files owned by ''root'' are exempted.)

In our environment, the management of a workstation (in the traditional sense of UNIX system management) is in the hands of the user currently using it. We make no secret of the root password on our workstations, as we realize that a truly unfriendly user can break in by the very fact that s/he is sitting in the same physical location as the machine and has access to all console functions. Therefore we cannot truly trust our workstations in the NFS interpretation of trust. To allow proper access controls in our environment we had to make some modifications to the base NFS software, and integrate *Kerberos* into the scheme.

Unmodified NFS

In the implementation of NFS that we started with (from the University of Wisconsin), authentication was provided in the form of a piece of data included in each NFS request (called a ''credential'' in NFS terminology). This credential contains information about the unique user identifier (UID) of the requester and a list of the group identifiers (GIDs) of the requester's membership. This information is then used by the NFS server for access checking. The difference between a trusted and a non-trusted workstation is whether or not its credentials are accepted by the NFS server.[12]

Modified NFS

In our environment, NFS servers must accept credentials from a workstation if and only if the credentials indicate the UID of the workstation's user, and no other.

One obvious solution would be to change the nature of credentials from mere indicators of UID and GIDs to full blown *Kerberos* authenticated data. However a significant performance penalty would be paid if this solution were adopted. Credentials are exchanged on every NFS operation including all disk read and write activities. Including a *Kerberos* authentication on each disk transaction would add a fair number of full-blown encryptions (done in software) per transaction and, according to our envelope calculations, would have delivered unacceptable performance. (It would also have required placing the *Kerberos* library routines in the kernel address space.)

We needed a hybrid approach, described below. The basic idea is to have the NFS server map credentials received from client workstations, to a valid (and possibly different) credential on the server system. This mapping is performed in the server's kernel on each NFS transaction and is setup at ''mount'' time by a user-level process that engages in *Kerberos*-moderated authentication prior to establishing a valid kernel credential mapping.

To implement this we added a new system call to the kernel (required only on server systems, not on client systems) that provides for the control of the mapping function that maps incoming credentials from client workstations to credentials valid for use on the server (if any). The basic mapping function maps the tuple:

<CLIENT–IP–ADDRESS, UID–ON–CLIENT>

to a valid NFS credential on the server system. The CLIENT–IP–ADDRESS is extracted from the NFS request packet and the UID–ON–CLIENT is extracted from the credential supplied by the client system. Note: all information in the client-generated credential except the UID–ON–CLIENT is discarded.

If no mapping exists, the server reacts in one of two ways, depending it is configured. In our friendly configuration we default the unmappable requests into the credentials for the user ''nobody'' who has no privileged access and has a unique UID. Unfriendly servers return an NFS access error when no valid mapping can be found for an incoming NFS credential.

Our new system call is used to add and delete entries from the kernel resident map. It also provides the ability to flush all entries that map to a specific UID on the server system, or flush all entries from a given CLIENT–IP–ADDRESS.

We modified the mount daemon (which handles NFS mount requests on server systems) to accept a new transaction type, the *Kerberos* authentication mapping request. Basically, as part of the mounting process, the client system provides a *Kerberos* authenticator along with an indication of her/his UID–ON–CLIENT (encrypted in the *Kerberos* authenticator) on the workstation. The server's mount daemon converts the *Kerberos* principal name into a local username. This username is then looked up in a special file to yield the user's UID and GIDs list. For efficiency, this file is a *ndbm* database file with the username as the key. From this information, an NFS credential is constructed and handed to the kernel as the valid mapping of the <CLIENT–IP–ADDRESS, CLIENT–UID> tuple for this request.

At unmount time a request is sent to the mount daemon to remove the previously added mapping from the kernel. It is also possible to send a request at logout time to invalidate all mapping for the current user on the server in question, thus cleaning up any remaining mappings that exist (though they shouldn't) before the workstation is made available for the next user.

Security Implications of the Modified NFS

This implementation is not completely secure. For starters, user data is still sent across the network in an unencrypted, and therefore interceptable, form. The low-level, per-transaction authentication is based on a <CLIENT–IP–ADDRESS, CLIENT–UID> pair provided unencrypted in the request packet. This information could be forged and thus security compromised. However, it should be noted that only while a user is actively using her/his files (i.e., while logged in) are valid mappings in place and therefore this form of attack is limited to when the user in question is logged in. When a user is not logged in, no amount of IP address forgery will permit unauthorized access to her/his files.

References

1. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Section E.2.1: Kerberos Authentication and Authorization System*, M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).
2. E. Balkovich, S. R. Lerman, and R. P. Parmelee, ''Computing in Higher Education: The Athena Experience,'' *Communications of the ACM* **28**(11), pp. 1214-1224, ACM (November, 1985).
3. R. M. Needham and M. D. Schroeder, ''Using Encryption for Authentication in Large Networks of Computers,'' *Communications of the ACM* **21**(12), pp. 993-999 (December, 1978).
4. V. L. Voydock and S. T. Kent, ''Security Mechanisms in High-Level Network Protocols,'' *Computing Surveys* **15**(2), ACM (June 1983).
5. National Bureau of Standards, ''Data

Encryption Standard,'' Federal Information Processing Standards Publication 46, Government Printing Office, Washington, D.C. (1977).

6. S. P. Dyer, ''Hesiod,'' in *Usenix Conference Proceedings* (Winter, 1988).

7. W. J. Bryant, *Kerberos Programmer's Tutorial,* M.I.T. Project Athena (In preparation).

8. W. J. Bryant, *Kerberos Administrator's Manual,* M.I.T. Project Athena (In preparation).

9. G. W. Treese, ''Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD,'' in *Usenix Conference Proceedings* (Winter, 1988).

10. C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, ''The Zephyr Notification System,'' in *Usenix Conference Proceedings* (Winter, 1988).

11. M. A. Rosenstein, D. E. Geer, and P. J. Levine, in *Usenix Conference Proceedings* (Winter, 1988).

12. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, ''Design and Implementation of the Sun Network Filesystem,'' in *Usenix Conference Proceedings* (Summer, 1985).