-- --

CITI Technical Report 93−1

Long Running Jobs in an Authenticated Environment

A. D. Rubin
rubin@citi.umich.edu

P. Honeyman
honey@citi.umich.edu

ABSTRACT

Current authentication systems require that a user have a valid token or ticket for a job to run. These tickets are issued with limited lifetimes, and their renewal requires a user to enter her password. We have developed a system called `lat` with which a user may schedule a batch job to be run at a later date in the current environment. The batch job is stored on a secure machine, and sent and received only in encrypted form. When it is time for the job to run, the server generates a ticket for the original user and sends it (encrypted) to the machine on which the job will run. The user is given an option to specify that tickets should be continually generated for the job until its execution has completed.

March 29, 1993

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Long Running Jobs in an Authenticated Environment

A. D. Rubin
rubin@citi.umich.edu

P. Honeyman
honey@citi.umich.edu

1. Introduction

Adaptations of Needham and Schroeder's authentication system [1] are a boon for establishing secure services in distributed systems. One such adaptation is the Kerberos Authentication system [2] of MIT's Project Athena. An unfortunate byproduct of building Kerberos-based systems is a loss of functionality, such as long running jobs. In this paper, we address this weakness and offer a solution.

Before Kerberos, UNIX authentication was coterminus with a login session. In the Kerberos system, tickets[1] expire, so that a compromised[2] ticket does not allow an imposter to masquerade as an authenticated user forever. Consequently, users are forced to reauthenticate on a regular basis, usually about once a day, to acquire fresh tickets.

For a Kerberos user to submit and successfully execute a long-running batch job that employs secure system resources, she must either physically reauthenticate whenever tickets are about to expire, or enter a password into a shell script. Similarly, it is impossible to schedule a batch job to run at a distant future date and be authenticated as the user.

This problem has been recognized as a difficult one. Lampson *et al.* [3] state that ‘‘It is a tricky exercise in balancing the demands of convenience, availability, and security.’’ They further state that ‘‘the basic idea is to have a single highly available agent for the user that replaces the login workstation and refreshes credentials for long-running jobs.’’ This approach is similar to the one we take in solving the problem for Kerberos-based systems.

1 Kerberos credentials.
2 *E.g.,* stolen.

2. LAT: A New Kerberos Service

This section describes a new service: `lat`, based on the UNIX `at` command. In a later section, we critique the design and offer suggestions for improvement.

Like `at`, which is used to schedule batch jobs at a specific time and date, `lat` offers a batch service. The principal difference between the two is that `lat` provides continuous Kerberos authentication to the batch job while it runs.

2.1. Overview of LAT

When a user wishes to schedule a batch job, she invokes the `lat` command with a syntax very similar to the UNIX `at` command. A spool file for the batch job is created, containing, among other things, the user's name, her current working directory, and her shell environment. So far, this is identical to `at`.

Next, `lat` requests a ticket for the `lat` service from the Kerberos ticket granting server on behalf of the user. This step is shown in Figure 1.

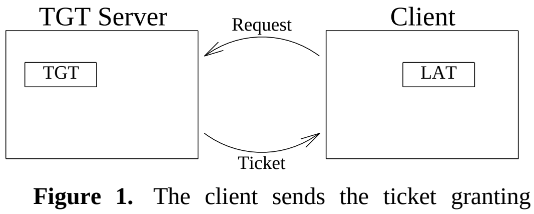

Figure 1. The client sends the ticket granting server a a request for a `lat` ticket. The request includes a ticket granting ticket (TGT) which proves the identity of the client, and enables the server to send the `lat` ticket to the client.

The `lat` ticket is then sent to a `latd` server. The server must run on a secure machine. The client and server then mutally authenticate [2], as shown in Figure 2.

After mutual authentication, the client and server

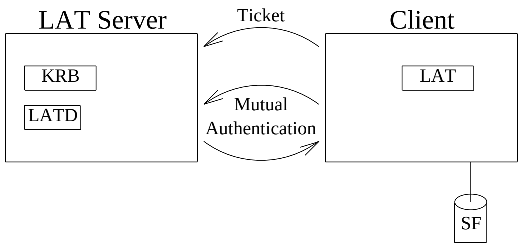

Figure 2. The client has generated a spoolfile (SF) for the `lat` job, which it stores on the local disk. It sends a `lat` ticket to the server (LATD), and the client and server mutually authenticate. Note that the Kerberos server (KRB) is running on the same machine as `latd.`

have available a shared DES [4] key, which we denote *K*, as shown in Figure 3.

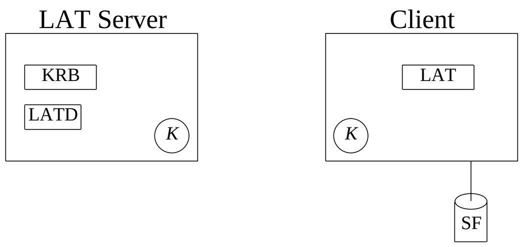

Figure 3. Initially, the spool file resides on the client machine. The client and server share a secret key, *K*.

The `lat` client uses *K* to seal the spool file. This hides the details of the batch request from prying eyes, as well as assuring its integrity. The encrypted spool file is then sent to the `latd` server, as shown in Figure 4.

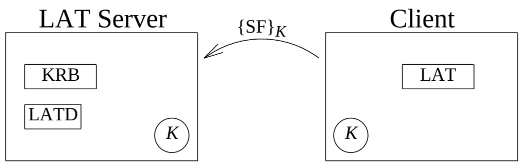

Figure 4. The client sends the spool file to the server, sealed under the session key, *K*.

Along with the spool file, the client sends information such as the time and date for the job to run, the user's environment, *etc*. The `latd` server receives the spool file from the client, unseals it, and stores the file away until it is time for the job to run. At this point, the client discards its copy of the spool file and listens to a well known port for activation. For reasons discussed later, the `latd` server runs on a Kerberos master or slave machine. This blocked configuration is depicted in Figure 5.

Periodically,[3] the server wakes up and checks to

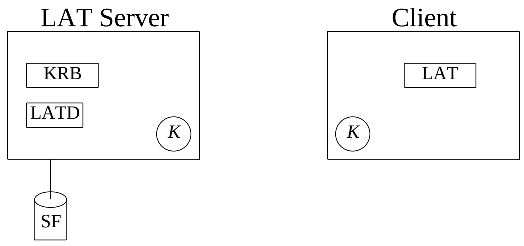

Figure 5. Figure The spool file is stored on the secure server machine, and is purged from the client machine.

see if any job is scheduled to be run. When the time for the job to run arrives, the server uses the Kerberos database to construct a ticket for the user. It then seals this ticket along with the spool file, and sends them back to the client machine, as shown in Figure 6.

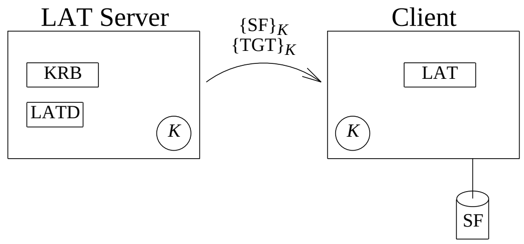

Figure 6. When it is time for the job to run, the encrypted spool file and a ticket granting ticket (TGT) are sent to the client under the session key.

Now the client can use the TGT to obtain tickets for other services. If the job will run for longer than the life of the ticket, or if the user suspects this may be the case, `lat` offers an option to renew tickets, in which case the server sends new tickets to the client as long as the job is running. Of course, care is taken to ensure that the job is still running. (This proves to be a difficult problem.)

After the job terminates, the spool file is removed from the client machine, and the `lat` process exits. A message is then sent to the server machine so that the `latd` process can exit too. Details follow in Section 3.6.

2.2. Implementation of the `latd` server

The server program, `latd`, runs as root. Moreover, the server machine contains a master or slave copy of the Kerberos database; any program running on such a machine must be trustworthy. The `lat` program runs on the client's host machine. It also runs as root, but care is taken to make sure that the user's job is not able to obtain a higher privilege than it should. The program

3 Once a minute, in our implementation.

uses the UNIX `setuid` facility to ensure that the user's batch job runs as that user. However, the actual `lat` program runs as root because it must perform operations that require special privileges, such as setting group IDs, deleting files, and copying files to and from the local disk.

When the `latd` server needs to issue tickets for a user, it sends a request to the Kerberos server. The Kerberos server returns a user ticket encrypted under that user's secret key. Unfortunately, the client machine does not necessarily have a user authenticated to it when this happens, so the user's secret key may not be available. Thus, the user's ticket issued by the Kerberos server is not readable.

To decipher the user ticket, the `latd` server uses the Kerberos database to access the user's secret key. Since user keys are stored encrypted under a master key in the Kerberos database, `latd` must first use the Kerberos master key. Then, the user's key is decrypted. Once `latd` has decrypted the user's secret key, it decrypts the user ticket received from the Kerberos server. Finally, `latd` changes the client address in the ticket to that of the target host that will run the job. The ticket is then ready to be sent to the user's host machine. The encryption of the ticket before it is sent over the network is the topic of Section 3.5.6.

It should be noted that for the ticket received from the Kerberos server to be readable, `latd` has to access the Kerberos database and the Kerberos master key. Thus, it is a requirement that `latd` run on a Kerberos master or slave machine.

When a user invokes the `lat` server, the ticket granting ticket is sent to the ticket granting server with a request for a `lat` ticket. After mutual authentication, the client receives a ticket and can then communicate with the server. Thus, `lat` behaves as any other Kerberos service.

3. The Theory Behind LAT

It seems rather contradictory to provide authentication for an absent principal. Authentication, by definition, means that a principal proves her identity, a very difficult task if she is no longer present. Thus, to schedule a long-running job, or one to be run at a later date, a principal must leave something around so that possession of this thing is equivalent to an authentication for that principal. A similar idea, called delegation is discussed by Lampson *et al.* in [3]. The authors define the ‘‘speaks for’’ relationship and provide rigorous definitions and proofs based on a set of axioms they define in the paper.

Ideally, we would like the user's batch job to delegate authority to the workstation, saying that the workstation speaks for the user. In general, though, we are dealing with the domain of untrusted workstations. Many workstations reside in public sites where many different users have access to them at all times. It is a fundamental assumption that nothing on such a workstation can be trusted. However, some compromises must be made to provide for authenticated long-running jobs; we elaborate on this theme in the next section.

3.1. The authentication problem for a vacant workstation

We call a workstation *vacant* whenever a given user's task must be run there while that user is not logged in. In the previous section, we assumed that nothing on the workstation could be trusted. The reason for this is straightforward: we must allow for the possibility that a user might obtain root privileges, *e.g.*, by booting the machine into single-user mode, whereupon the privileged user might replace any or all utilities on the workstation, including the operating system image itself. The only objects on a public workstation safe from such attack are those that are encrypted. Yet, we take it as given that the encryption key may not reside on the workstation, even if well hidden. It has long been agreed by experts that ‘‘security through obscurity’’ should never be relied on for system secuity, *e.g.*, Kahn [5] cites Kerckhoffs' classic treatise on military security [6]; a more modern view is espoused by Saltzer and Schroeder [7] in describing ‘‘open design’’ as one of the basic principles of information protection:

> The design should not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords. This decoupling of protection mechanisms from protection keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards. In addition, any skeptical user may be allowed to convince himself that the system he is about to use is adequate for his purpose. Finally, it is simply not realistic to attempt to maintain secrecy for any system which receives wide distribution.

Voydock and Kent amplify this perspective: ‘‘data encryption is the fundamental technique on which all communications security measures are based'' [8]. Any techniques or protocols may as well be open and known if there is nothing gained from hiding them.

Therefore, in a secure, distributed authentication system, data must travel across the network encrypted; for two peers to communicate this way, they must share a secret. Thus, the user must place something on the workstation which the server can later use for mutual authenticatation.

Lampson *et al.* describe a mechanism whereby a vacant workstation could share a secret [3]. Their method requires that a machine possess a private key stored in nonvolatile memory. In addition to the private key, certificates and other rules must be stored on the boot ROM.

Aside from the fact that our workstations do not contain this information in ROM, Lampson's method requires a public key system, which is not compatible with Kerberos. At some future date, it may be possible to authenticate a workstation, whereupon it will not be necessary for the client to leave anything on the machine.

3.2. How LAT authenticates to a vacant workstation

For the server to send an encrypted spool file and tickets back to the client's host machine, some shared secret must be left on the workstation. The creation and responsibilty of this secret is illustrated in Figure 7. To leave this secret on the workstation, a new random key is generated, which we denote *N*. *K*, the `lat` session key, is encrypted with *N* and stored on the client machine. *N* is then sent encrypted under *K* to the server. Note the symmetry here: the client holds $\{K\}_N$ but sends the server $\{N\}_K$.

After sending the spool file and other information to the server, the client erases all of this information from disk and from memory. All that the workstation keeps is $\{K\}_N$, the session key from the original `lat` ticket encrypted under *N*, maintained in the address space of the `lat` process on the client machine.

When it is time for the server to send the spool file and tickets to the workstation, it encrypts using *K*, so that the workstation can decrypt them. The server first sends *N* to the workstation in the clear. The workstation then unseals the original session key, and thus decrypts the spool file and tickets when they arrive.

It should be noted that by itself, eavesdropping on the network does no harm: the only junction served by *N* is to unseal the key in the process memory on the workstation. Once the session key is unsealed, the spool file and tickets sent across the network can be decrypted by the `lat` agent sitting on the workstation.

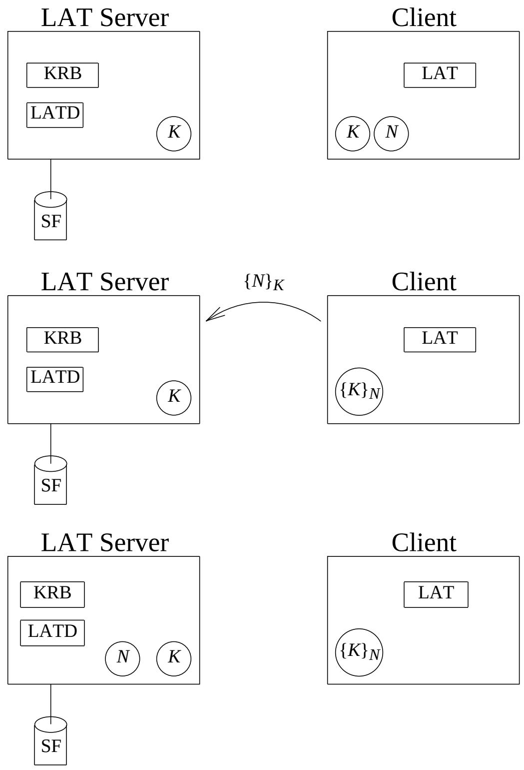

Figure 7: How *K* is secured on the client. The first step is illustrated in the top diagram. The client generates a random key, *N*. Then, as shown in the next diagram, *N* is used to encrypt the session key, *K*, which remains on the client machine. Then, *N* is sent to the server under the sesssion key. Finally, as shown in the bottom diagram, *K* and *N* are stored on the server. When it is time for the job to run, *N* is sent to the client to unseal $\{K\}_N$.

3.3. Risks of running LAT with a vacant workstation

In this section, we analyze the risks involved in running `lat` on a vacant workstation. If the workstation is rebooted, then the process memory is lost. Although a denial of service results, this can be reported back to the user and no real harm is done. If an imposter manages to gain control of the machine without erasing the memory, and examines memory to find the secret key, this will give no advantage, since the secret key is encrypted with *N*. In fact, a few other safeguards are in place. The imposter has no idea when a job is scheduled to run, since all such information has been kept secret and no longer resides on the workstation.

To compromise a job, an imposter must be on the workstation when *N* arrives, and have already acquired the session key, encrypted under *N*, from the process memory. The imposter must then happen to be eavesdropping when *N* is sent, and then be able to decrypt the spool file and insert bogus commands. However, if the imposter has this capability, then the imposter has completely compromised the workstation and would be able to interfere with the job, which must run in memory.

To deny service, an imposter can simply reboot a workstation, but `lat` can notify the user of unsuccessful batch jobs (soon to be implemented). As long as users understand this risk, they can choose whether to use the `lat` service. `Lat` does not compromise the security of people who do not use it.

3.4. The single user approach

The vacant workstation problem was addressed by Treese at MIT, where access to an Athena workstation is limited to one user at a time [9]. Treese reports that ''experience has shown that this is an acceptable limitation.'' Placed in our context, workstations could prohibit any login while any `lat` job is scheduled. This would make `lat` client machines much more secure. However, this could result in a serious denial of service. A malicious user could schedule a `lat` job which would monopolize a workstation for any amount of time. Also, a user could log in and prevent a scheduled `lat` job from running. Both these forms of denial of service can be detected. In fact, `latd` could maintain a database of pending `lat` jobs, and any abuse of the system could be easily traced to the offending user.

3.5. Generating tickets for a user

When it is time for a batch job to run, the `lat` server must obtain a ticket for the user. It cannot simply issue a request for a ticket the way a user does because this request looks up the address from which it comes, and puts the client `IP` address into the ticket. Thus, the ticket must be constructed manually by the server. The steps taken by the server are as follows:

- Get the master key for Kerberos
- Get the TGT secret key from the Kerberos database
- Decrypt the TGT key with the master key
- Create a TGT ticket for the user
- Zero out the master key and other keys from memory

A brief discussion of each of these steps follows. This discussion makes it clear that the `lat` server must have the Kerberos database available to it, and must either run on the same machine as Kerberos, or on a Kerberos slave machine.

3.5.1. Get the master key for Kerberos

The following steps are taken to get the master Kerberos key. We call `gettimeofday` to set the Kerberos time. Then we call routines to get and verify the master key. After that, the version number is checked. If the wrong version number appears, or if `kdb_verify_master_key` returns an error, we log the error and exit. In this case, no tickets can be generated, and mail is sent to the user (mail not implemented yet).

3.5.2. Get the TGT secret key from the Kerberos database

To get the TGT secret key from the Kerberos database, we call `check_princ,` which checks for expiration times on the master key and the service. It also fills the `Principal` data structure with information containing the key (encrypted under the master key), realm, *etc*.

3.5.3. Decrypt the TGT key with the master key

This step is straightforward. We use the master key to decrypt the TGT key.

3.5.4. Create a TGT ticket for the user

We have the TGT ticket that will be used to encrypt the ticket once it is constructed. Inside the ticket, we place user information such as name, instance, and realm, obtained from the call to `check_princ.` Then, we add the client host address that we obtained with a call to `getpeername` to see who the client was. In addition, we generate a random session key and include that in the ticket. Other usual information such as the time and ticket lifetime are also included. A lifetime of about 25.5 hours was chosen here, but that was arbitrary and based on the choice observed in existing Kerberos services. The ability to specify the lifetime could be added later as a user-specified option to `lat.`

3.5.5. Zero out the master key and other keys from memory

Finally, we zero out all of the memory we used for highly sensitive information such as the master key. Even though this is a trusted machine, it never hurts to take added precautions.

3.5.6. Send the ticket to the client

Once the ticket has been constructed properly, it is encrypted under the session key available to the client and sent across the network. The client decrypts the ticket, and stores it with the batch job user as the owner, with no permissions for anyone else.

3.6. Renewing the tickets

If a batch job is to run for more than the lifetime of a TGT ticket, then the workstation must receive a new ticket for the user. The ability to renew tickets is added as a command line option. If a user prefers for jobs which outlive the tickets to die rather than have tickets generated until the job exits, she can chose to omit this option.

The `lat` program forks a master process to run the batch job. Before the job is actually run, this process forks a sub-process. This sub-process is in charge of maintaining the user's authentication on the workstation. When the master processing the job completes, it terminates the sub-process, and sends a message to the server that it is finished. This message is not necessary, but it is sent so that the `latd` process on the server will exit normally. The details of this scheme follow.

The process that maintains the authentication works as follows. It goes to sleep until the authentication ticket is about to expire. Then, it calls `gettimeofday` to get the current time. The time is sent, encrypted, to the server, ensuring against replay, and proving possession of the secret key. The server checks the time, and if it matches to within a minute, is convinced that a new ticket must be sent. The server then constructs a new ticket for the user and sends it across the network. The workstation replaces its current ticket file with the new one, and then goes back to sleep.

The server on the other end waits for a request for tickets or a message from the client that the job has terminated. When a request comes in, the server checks that the time is within a minute of the current time. If not, the request is ignored, and no tickets are generated. If the time is correct, then the user creates a new TGT ticket for the user, encrypts it, sends it to the client, and continues waiting for requests.

4. Conclusions

Using `lat,` it is now possible to run a batch job with authentication without manually renewing user tickets. The risks of having valid tickets on a vacant workstation are inherent to `lat.` To minimize the risk, the secret key used to authenticate to the server when the user is gone is maintained, encrypted in the process memory of the `lat` process on the client machine.

The `lat` server runs on a secure machine with access to the Kerberos database. It uses this database to generate tickets for users and the TGT service. Everything sent across the network is encrypted first with the secret key available on the workstation.

When scheduling jobs to run on a vacant workstation, there must be some security compromise for authentication to take place. This problem will remain until there is some way to actually authenticate a workstation.

5. Future Work

We are considering some enhancements for future implementation.

Some utilities could be added to `lat.` In particular, we are working on three: `latq, latrm,` and `latconts. Latq` will display the queue of impending jobs for a user. `Latrm` will allow a user to cancel a job by removing it from the queue. And `latconts` will display the contents of a scheduled job. These utilities are being Kerberised with authentication and encryption protocols.

Another possibility for future work is to add flexibility to the `lat` service. For example, a user may wish to specify that a workstation should only maintain valid tickets between certain hours, when presumably, she believes it is safer. An option can be added in which the user specifies the lifetime of tickets, and perhaps provides some conditions under which they should or should not be renewed.

Another feature which could be added to `lat` is the ability for a process to save its state before a ticket expires, send it (encrypted) to the server, and then wait until the user reauthenticates to continue running. This feature would be useful in a case where the workstation somehow realizes it has been compromised, or is about to be.

Acknowledgements

The first author thanks David Snearline and Scott Dawson for their continuing help with Kerberos and programming details, Dan Muntz for his patience and help with the working environment at CITI, Michael Stolarchuk for his support early on in the project, and Bernie Galler for his constant motivation and support.

References

1. 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).
2. J.G. Steiner, B.C. Neuman, and J.I. Schiller, ‘‘Kerberos: An Authentication Service for Open Network Systems,’’ pp. 191–202 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).
3. B. Lampson, M. Abadi, M. Burrows, and E. Wobber, ‘‘Authentication in Distributed Systems: Theory and Practice,’’ *ACM Transactions on Computer Systems* **10**(4) (November, 1992).
4. National Bureau of Standards, ‘‘Data Encryption Standard.,’’ *Federal Information Processing Standards Publication*(46) (1977).
5. D. Kahn, *The Codebreakers*, Macmillan Publishing Co., New York (1967).
6. A. Kerckhoffs, *La Cryptographie Militaire*, Libraire Militaire de L. Baudoin & Cie., Paris (1883).
7. J.H. Saltzer and M.D. Schroeder, ‘‘The Protection of Information in Computer Systems,’’ *Proc. of the IEEE* **63**(9), pp. 1278–1307 (September, 1975).
8. V.L. Voydock and S.T. Kent, ‘‘Security Mechanisms in High-Level Network Protocols,’’ *Computing Surveys* **15**(2) (June, 1983).
9. G.W. Treese, ‘‘Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD,’’ *USENIX Winter Conference, Dallas Texas*, pp. 175–182 (February, 1988).