Workstation Services and Kerberos Authentication at Project Athena

Don Davis, MIT Staff
Ralph Swick, Digital Equipment Corp.
03/17/89

Introduction

This document proposes solutions for two problems obstructing Project Athena's implementation of workstation services.

The principal problem is that workstation services demand a more flexible mutual-authentication protocol than Kerberos currently provides. The egregious X access-control hack, xhost, for example, has lack of authentication as its root cause. This protocol weakness is also the reason that public workstations can't accept authenticated connections from rlogin, rcp, rsh, etc. We propose an extension to the Kerberos Ticket Granting Service protocol, that cleanly supports user-to-user mutual authentication.

Our second proposal addresses the problem of ticket propagation. Currently, if a user wants tickets that are valid on a remote host, he has to run kinit in an encrypted rlogin session, unless he's willing to send his password in cleartext. As an example of the use of our protocol extension, we describe a Kerberos application that would support a limited facility for secure ticket-propagation.

Authentication of Workstation Services

Problem to be Solved

Public workstation users can't offer authenticated network services. Currently, only physically secure hosts can offer such services, because Kerberos' client-to-server authentication requires each server to store its private key locally. Public workstations are insecure, so we can't extend this approach to workstations' services.

The basic Kerberos protocol,[1] which allows a user to gain a service ticket in exchange for a password, is not at fault in this problem. In fact, the basic protocol, lacking the complications of TGTs and *srvtab*, offers a trivial, albeit limited, solution: Kerberos can supply anyone who asks with an encrypted key/ticket pair of the form $\{K_{c,sp}$,..., $\{ T_{c,sp}\} K_{sp}\} K_c$.[2] Of course, the keys K_c and K_{sp} are private keys, so both client and server must enter their passwords each time they make a connection. The problem, restated, is thus to relieve users of the frequent need to enter their passwords.

The clients' half of this problem has been completely solved by the Ticket-Granting-Ticket (TGT) protocol. Athena has addressed the servers' half of the problem, but only weakly, by storing each server's private key in *srvtab*. Thus, clients and servers currently use *user-to-host* authentication. This doesn't work on public workstations, for two reasons:

[1]Needham & Schroeder's 1978 protocol, plus timestamps. See: Roger M. Needham and Michael D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers". CACM Vol. 21, No. 12, Dec. 1978, pp. 993-999.

[2]Here, the subscripts "c" & "sp" refer to the client and service-provider, respectively. The ellipsis represents our omission of the timestamp, server's ID, and other data. Otherwise, our notation follows Steiner, Neuman, & Schiller, "Kerberos: An Authentication Service for Open Network Systems", USENIX Winter Conference, February, 1988.

- Public workstations are vulnerable to various privacy attacks, and hence cannot securely hold any long-lived secret.
- In most cases, what we really want is user-to-user, not user-to-host, authentication.

An immediate corollary of public workstations' insecurity is that idle public workstations' services cannot be authenticated, because insecure hosts can readily be impersonated in any protocol. Thus, we believe that no general user-to-host scheme can embrace public workstations. Accordingly, this proposal will address only the goal of a fully general user-to-user mutual authentication protocol. A user-to-user protocol raises the problem of dynamically mapping users to hosts, but we will not address such mapping in this document.

Constraints on Solutions

These are non-negotiable, in case you're wondering:

1. We can't add state to the Kerberos server's process at all.
2. We can't add frequently-changing state to the Kerberos database.
3. We should make at most one transaction with Kerberos per connection.
4. More generally, loading the Kerberos server is always to be avoided.
5. No infinite-life keys (like *srvtab*'s) can be stored on an insecure host (for example, public workstations).

Constraints 1 - 4 are "scaling issues". Constraints 1 and 2 limit the difficulty of replicating Kerberos with slave servers. As a consequence of Constraint 1, Kerberos can never initiate any protocol, because to ask for something requires that Kerberos await the response, which requires process-state. For the same reason, Kerberos can't measure time intervals at all.

Discussion of the Problem

If the server workstation is to autonomously authenticate on its user's behalf, it will have to store a secret that only the user and Kerberos share; this is axiomatic. Furthermore, because the workstation's secret could be compromised at any time,[3] this secret must be short-lived. We propose to use the user's session-key with the TGS as the "service secret".

Then Kerberos' most natural response to a service-ticket request takes the form $\{ K_{c,sp}$,..., $\{ T_{c,sp}\} K_{sp,tgs}\} K_{c,tgs}$. Unhappily, it is not straightforward to enable Kerberos to build such a response. If Kerberos is to use both users' TGS session keys to encrypt the service-ticket, Kerberos must receive both users' TGTs[4] simultaneously. Note that Constraint 1 implies that Kerberos cannot judge "simultaneity" of these tickets' arrival, unless they arrive together in one message.

---

[3]via an unattended console, for example.

[4]To a good approximation, C's TGT = $\{ C, \text{time/life}, K_{c,tgs}\} K_{tgs}$.

It is troublesome, though, for one user to pass both TGTs to Kerberos, because the TGT protocol requires that each TGT be presented to Kerberos with a time-stamped authenticator. Further, the TGT protocol has no provision for one user to present another user's credentials. However, for one user to possess another's TGT is actually neither troublesome nor remarkable, since in order to use the TGT, any impersonator would need the corresponding session key. Indeed, when any user requests service tickets, he sends his TGT along in a cleartext request, making the TGT available to anyone on the net.

The source of the TGT protocol's "crossed-credentials" prohibition, is a flawed analogy between TGTs and service tickets. The basic Kerberos protocol requires that a user present an authenticator when using his service ticket, so as to prevent replay of service tickets. The TGT protocol conservatively makes the same requirement, on the assumption that what is secure for other services, is secure for the ticket granting service. But, in fact, a TGT-mediated service ticket request is actually more analogous to the basic Kerberos ticket-request, which does not include an authenticator: neither request can usefully be replayed, because the TGS' responses are always encrypted in the requester's key.

Thus, in essence, a principal authenticates himself by using his secret key; reading an encrypted message serves this purpose as well as does sending an encrypted authenticator, and doing both is redundant. Further, Kerberos' role is not to authenticate the service's principals, but to enable the principals to authenticate one another. Thus, we argue that the TGT-protocol's authenticator requirement can safely be relaxed, so as to allow either member of a client-server pair to present both members' TGTs.[5]

Notation

We introduce the notation $t_{c,sp}$ for the conversation key $K_{c,sp}$, service-id, ticket lifetime, and other data, that accompany the service ticket in a credentials message from Kerberos. The identity $T_{c,sp}$ == (C, $t_{c,sp}$) is a good approximation.[6] Thus, what we've represented as

$$\{ K_{c,sp} \;, \ldots, \{ T_{c,sp} \} K_{sp,tgs} \} K_{c,tgs}$$

is properly written

$$\{ t_{c,sp} \;, \{ T_{c,sp} \} K_{sp,tgs} \} K_{c,tgs}$$

As mentioned above, our notation otherwise follows Steiner, Neuman, and Schiller [88].

[5]Actually, the TGS protocol could retain the authenticator requirement, if the TGS were willing to unseal TGT_{sp} after verifying the credentials in TGT_c. This would preserve some accountabilty for one kind of service-denial attack.

[6]The Kerberos Request For Comments (RFC), currently in preparation, is the best reference for the existing protocol's message contents.

Our Proposed Solution

We've chosen to have the client do the talking with Kerberos, because to do so requires time-out state, which burden can't be borne by all application servers. An added benefit of this choice, is that if a network connectivity fault separates a server from Kerberos, some of its clients will still be able to authenticate.

1. Client C asks server SP for service, in cleartext.

   C ----> SP : (C wants SP)

2. SP sends its TGT, but not its session-key, to C.

   SP ----> C : $\{ T_{sp,tgs} \} K_{tgs}$

3. C asks Kerberos for a service ticket, sending SP's TGT and C's own TGT.

   C ----> Krb : $(\{ T_{c,tgs} \} K_{tgs} , \{ T_{sp,tgs} \} K_{tgs})$

4. In response, Kerberos:
   - decrypts the two TGTs, yielding the users' names and TGS session keys;
   - prepares a new session key $K_{c,sp}$ for C and SP to share;
   - composes service-ticket contents $T_{c,sp}$ from the TGTs' name-fields, the new session key, and other data;
   - uses SP's TGS session key $K_{sp,tgs}$ to encrypt the ticket contents into a service ticket;
   - sends the service ticket, the new session key, and other credentials to C, encrypted in C's TGS session key $K_{c,tgs}$.

   Krb ----> C : $\{ t_{c,sp} , \{ T_{c,sp} \} K_{sp,tgs} \} K_{c,tgs}$

5. On receipt of the ticket/key pair, C:
   - uses C's TGS session key to decrypt the credentials, yielding the new session key $K_{c,sp}$, the service ticket, and other data;
   - checks the service-provider's name and the timestamp in $t_{c,sp}$,
   - uses $K_{c,sp}$ to encrypt an authenticator, and
   - sends the service-ticket and authenticator to SP.

   C ----> SP : $(\{ Auth_c \} K_{c,sp} , \{ T_{c,sp} \} K_{sp,tgs})$

6. On receipt of the ticket-authenticator pair, SP:
   - uses SP's TGS session key to decrypt the ticket, gaining $K_{c,sp}$;
   - checks the names and the lifetime in $T_{c,sp}$,
   - uses $K_{c,sp}$ to decrypt C's authenticator, and
   - uses $K_{c,sp}$ to encrypt a corresponding authenticator of its own, which it returns to C (optional for physically-secure services).

SP ----> C : { Auth_{sp}} $K_{c,sp}$

7. For each additional connection, C and SP need to repeat only messages 5 and 6 (optional).

Step 1 seeks to verify that SP is in fact available at the message's destination; since Kerberos will not be handling user-to-host *mapping*, such a query is probably desirable in any user-to-user authentication protocol. Step 2 serendipitously offers a solution to the difficult problem that a client can't distinguish in his ticket file between two identically-named service tickets: we propose that the SP's TGT is just the handle we need. Indeed, we propose that if two services have the same name and the same TGT, they *should* be indistinguishable.

Note also that in step 3, C specifies SP not by name, but by giving SP's TGT. In step 4, the TGS uses the TGTs' name-fields to build C's credentials, thereby securely identifying C and SP to one another as the owners of the key $K_{c,sp}$. Thus, C's and SP's checks of the credentials' name-fields foils intruders' replay of TGTs in the unauthenticated messages 2 and 3.

Note finally that Kerberos, to support this protocol, doesn't need access to the database, but needs only the TGS' service-key K_{tgs}. Thus, our proposed changes affect only Kerberos' Ticket Granting Service; the Kerberos database would not be changed.

Ticket Lifetimes and Renewal

The protocol we've presented so far, doesn't support ticket renewal. The service ticket is timestamped to expire as soon as either principal's TGT expires.[7] Whenever either user runs kinit to refresh her TGT, the client and service-provider processes need to be able to renew their conversation key and service ticket. This renewal of session credentials should proceed invisibly to the users.

There are three expiration/renewal scenarios:

- Servers' right to accept connections should expire with their TGTs; all remaining clients' service-tickets will expire simultaneously. These clients should renew their service-tickets only when they need a fresh connection.
- Clients' right to use a conversation key in an established service-connection may expire, if the service applies the service-ticket lifetime to the conversation key.
- Established client/server sessions may wish to change their conversation keys periodically, even if the service-ticket doesn't expire.

Service-ticket lifetime enforcement must be coded into the application-servers, as is done now. Clients should not try to enforce anticipated lifetimes on tickets, because servers may have idiosyncratic lifetime-rules. Once the client realizes that it needs a new ticket/key pair, all three types of renewal require that the client talk again to Kerberos with up-to-date

[7]This assumes that the maximal service-ticket lifetime == TGT lifetime. These lifetimes may be different.

TGTs. Thus, in step 4, Kerberos should be able to respond to out-of-date TGTs with an error-code that tells which TGT has expired, so that the client-user can know what to do.

User-To-Host Authentication

Should this protocol supplant or supplement the existing protocol? The main argument for grandfathering is that large-scale servers are typically secure, so they needn't bear the cost of the extra exchange 1 - 2. *A fortiori*, some services don't grant connections, but just want to accept authenticated messages, and therefore should use the briefest protocol possible.

We propose nevertheless to replace the existing user-to-host protocol with our protocol. Our main concern is that the Kerberos protocol should not become any more unweildy than it is already. Further, grandfathering the existing protocol will probably complicate programs like rlogin, which will need to use both user-to-user and user-to-host authentication.

A physically-secure server would still keep a host-principal private key in *srvtab*, but would use the key to get a TGT; its daemons would use the TGT in this protocol in order to accept connections. This arrangement is also necessary for insecure servers,[8] where administrators can't leave their personal TGTs unattended.

Hosts' TGTs should be non-expiring; otherwise, our protocol's uniformity comes at the cost of these hosts having to maintain up-to-date TGTs. After all, such long-lived session-keys wouldn't be any more vulnerable to cryptanalytic attack than *srvtab* keys are now. In the worst case, a Kerberos application can read *srvtab* to renew short-lived TGTs automatically.

Naming and Authorization Issues

A service-provider may wish to destroy his normal tickets before offering services to the network, so as to protect his client-identity from theft. Ideally, only unattended servers and cycle-servers (rlogin etc.) would need this precaution, but in principle, we shouldn't assume that even X servers and fingerds aren't providing more access than we expect. Thus, a service-provider should be able to enter our protocol with something other than a normal user TGT in hand. This section discusses the consequences of introducing *service instances* to Kerberos.

Lacking normal TGTs, a server should be able to enter our protocol with a service-instance TGT, in the name *username*.*service@realm*. The Kerberos protocol makes no restrictions on how many different instances a user may use to authenticate himself, but service-instances do present several problems:

1. Currently, the service-administrator will have to enter a separate password in order to gain each service-instance TGT.
2. The service-administrator will have no assurance that all network services will keep his service-instances' and client-instances' access separate.

[8]At MIT's Laboratory for Computer Science, one of the Kerberos beta-sites, *no* servers are physically secure.

3. With a server-instance available, some clients will have to decide at connect-time whether to use a service which lacks the access of a client instance.
4. If each user has a service-instance for each service he can offer, this proliferation will not merely add to the Kerberos Database's bulk, but will multiply it, probably by the number of workstation-services.

The service instance will act as a proxy for the service-administrator and for the client, because we anticipate that daemons will often need to provide "piggybacked services." For example, one can envision many services' needing their clients' X service (Note that piggybacked services generally shouldn't require that clients propagate their tickets). Proxies definitionally require an authorization mechanism. Here, because the service instance's TGT does double duty, the authorization needs of the client and service-administrator conflict. Specifically, the service-instance should not enjoy any of the service-administrator's client access, yet it must act as a client to serve its own clients. This conflict narrowly constrains our definition of a service instance's authorization.

To fully support piggybacked services, all application-protocols will have to restrict service-instances' authorization. Further, the usual access-control files will probably not suffice. We expect users' access-control lists to be quite volatile in the presence of workstation services, so that memory-cached lists will be necessary. This volatility will therefore probably require a sophisticated authorization library and some kind of centralized authorization support.[9] We do not propose to implement an authorization service soon.

If we disallow piggybacked workstation services, the service-instance need no longer be the clients' proxy, so much less central support can still relieve application-protocols of service-instance restrictions. It would suffice to allow a service-provider to spawn a single "weak" service-instance named, perhaps, *username*.**weak**@*realm*, which the TGS would reject as a client unauthorized for TGS. "Spawn" here, means that these weak instances would *not* appear in the Kerberos Database, but would gain TGTs on the strength of a normal instance's password or TGT.

An intergrade solution is possible as well, but is much less attractive: service instances would be spawned as above, but their instance-fields could have a variety of values, e.g., "X", "nfs", etc., which would be tabulated in an ancillary KDB. Further, these instances would be allowed to gain some service tickets, unlike weak instances; the TGS would regulate the service instances' access to service-tickets via its own access-control list.

In summary, we propose that weak instances may prove convenient, but probably aren't necessary. In the long run, some sort of authorization service will be necessary, since implementors will want to "piggyback" services. Until we implement one form or another of centralized authorization-support, we recommend that *no* service instances be created.

[9]Kerberos was originally named after Hades' three-headed watchdog because it was to provide not only authentication service, but authorization and accounting services, too. As the system developed, these auxiliary functions were deferred, on the grounds that authentication proved to be dauntingly subtle by itself.

Ticket Propagation

Problem to be Solved

Kerberos' current suite of applications doesn't allow users to get tickets for use on remote hosts. Rlogin users sometimes need such tickets in order to authenticate their remote sessions, e.g., so as to remotely access their NFS lockers.[10] We anticipate that other remote processes will need access to their client-users' tickets.

Constraints on Solutions

- Passwords and keys require encrypted transmission.
- The propagated tickets must be created anew for the recipient host; that is, the tickets' format must retain the "host id" field.
- Propagation of tickets must require a password; that is, it can't be automatic. (to prevent "unattended console" ticket-thefts).
- The local host must not cache tickets for a remote host (unattended console again).
- Propagated should normally have a reduced lifetime, since it's harder for the user to destroy them.
- As usual, we prefer not to change the Kerberos protocols.

Actually, it would probably be safe to allow automatic propagation of reduced-authorization tickets, but this is hampered by the difficulty of adding a notion of "reduced authorization" to Kerberos. Until it's better understood, automatic propagation is risky enough that Kerberos should only support it, when an application demonstrates an overriding need.

Discussion of the Problem

We propose a new service, called "rkinit", whose purpose is to transfer tickets on encrypted connections. How will ticket-propagation be used, and how will it work? Depending on whether the donor or the recipient initiates the transfer, we'll distinguish between "pushing" and "pulling" tickets, respectively. For example, a user might push tickets to a remote host before using rlogin, or he might rlogin first, and then use the remote session to pull his tickets after him.

Pulling is more convenient for rlogin and telnet users, who don't always need remote tickets. It would be nice to support both pushing and pulling of tickets at Athena, but only pushing is necessary. It's likely, in fact, that only rlogin and other "cycle services" can use pulling to advantage, so that it's best to equip those protocols with toggled-encryption. This would allow such users to run (r)kinit remotely and securely.

[10]Project Athena's NFS-implementation demands Kerberos authentication for protected accesses.

Our Proposed Solution

Pushing is the more elegant approach:

1. The donor requests rkinit service of the receiver host, and uses the resulting encrypted connection to identify himself.
2. The receiver rkinitd asks Kerberos for normal tickets in the normal way. Rkinitd then returns Kerberos' encrypted response to the donor.
3. The donor code prompts the user for his password, uses the password to decrypt the tickets just as kinit does, checks the tickets for freshness, and returns the tickets to the receiver, via the encrypted connection.
4. The receiver host's rkinitd puts the tickets into a ticket-file.

Pulling is quite hard to implement, because it always requires that the donor-user see a remote process' password-prompt.

1. The receiver runs kinit to get tickets, which are encrypted in the donor's private key. Kinit must be told the donor's host-name.
2. kinit then calls the donor's host, and uses the receiver-host's administrator's TGT to gain an encrypted connection.
3. The encrypted connection can be used to either get a password from the donor, or to send the tickets to him for decryption. In either case, the donor's host must raise a password-prompt somewhere. This is difficult if X-windows aren't available, and spoofable even then. After decrypting the tickets, the donor returns them to the receiver.
4. The receiver host's rkinitd puts the tickets into a ticket-file.

In summary, we propose that rkinit use the pushing protocol. In either case, rkinitd has to be careful to put the tickets into the correct ticket file, if multiple users/sessions are present. This is another aspect of Kerberos' naming problem.

Acknowledgments

Mark Lillibridge and Jon Rochlis helped us solve the problem. Jennifer Steiner suggested the problem statement, and answered many questions. Jerry Saltzer found a security hole in a late draft of this document. Cliff Neuman, Steve Miller, Mark Eichin, and Stan Zanarotti all offered trenchant insights, as we expected. Dan Geer read and understood at least six drafts of this document.

Appendix: Proof of Correctness for the Proposed Protocol

This proof uses a formal protocol-analysis logic.[11] We begin by breaking step 3 into two single-ticket messages, and analyze what happens when the TGS receives a single TGT:

Let Y_a = (A<--$K_{a,tgs}$-->TGS), N_a = (A, time, life),
and X_a = (N_a , Y_a, #(Y_a))
Then we're analyzing the message
C --> TGS : {X_a} K_{tgs}.
TGS |+ (Krb<--K_{tgs}-->TGS) and TGS <) {X_a}K_{tgs}
so TGS <) X_a and TGS |+ Krb |~ X_a, by msg-meaning rule.
now, the nonce N_a is principally a lifespan, so TGS |+ #(N_a),
and TGS |+ Krb |+ X_a, by nonce-verif. rule.
TGS |+ Krb |+ (Y_a, #(Y_a)); we assume that
TGS |+ Krb => (Y_a, #(Y_a)),
so TGS |+ Y_a and TGS |+ #(Y_a), by jurisdiction rule.

Substituting C & SP for A in X_a and Y_a, we find that these conclusions provide what we need to assume of the keys $K_{c,tgs}$ & $K_{sp,tgs}$.

The next protocol step is the credentials message:
Let Y = (C<--$K_{c,sp}$-->SP) and X = (Y, #(Y)), and
Z = (N_{sp} , X)
Then we're analyzing the message
TGS --> C: { Z, { C, Z} $K_{sp,tgs}$ } $K_{c,tgs}$.
we have C |+ (C<--$K_{c,tgs}$-->TGS), and C <) { Z,... } $K_{c,tgs}$, so
C |+ TGS |~ (Z, { C, Z} $K_{sp,tgs}$), by msg-meaning rule.
As above, C |+ #(N_{sp}), so C |+ #(Z, { C, Z}), and
C |+ TGS |+ (Z, { C, Z} $K_{sp,tgs}$), by nonce-verif. rule.
In particular, C |+ TGS |+ X, and we assume that C |+ TGS => X,
so we have C |+ X, so C |+ Y and C |+ #(Y), as desired.
Further, we have C <) { C, Z} $K_{sp,tgs}$.

Now we analyze the service request, with ticket & authenticator:
C --> SP: { C, Z} $K_{sp,tgs}$, ({ N_c, Y}$K_{c,sp}$*signed C*)
SP <) { C, Z} $K_{sp,tgs}$ and SP |+ (SP<-$K_{sp,tgs}$->TGS), so
SP |+ TGS |~ (C, Z). Now, recall that Z = (N_{sp}, X);
as usual, SP |+ #(N_{sp}), so SP |+ #(N_{sp}, X),
so SP |+ TGS |+ (N_{sp}, X), by the nonce-verif. rule.
In particular, SP |+ TGS |+ X; we assume SP |+ TGS => X, so SP |+ X.
That is, SP |+ Y and SP |+ #(Y), as desired.
Further, SP <) ({ N_c, Y}$K_{c,sp}$*signed C*),
so SP |+ C |~ (N_c, Y) and SP <) (N_c, Y).
SP |+ #(N_c), so SP |+ #(N_c, Y), so SP |+ C |+ (N_c, Y).
Thus, SP |+ C |+ Y.
Since we already have C |+ Y, this completes C's authentication to SP.

The analysis of SP's responding authenticator is analogous to
that of C's authenticator.

[11]Michael Burrows, Martin Abadi, and Roger Needham, "Authentication: A Practical Study in Belief and Action". (1987) Digital Equipment Corporation Systems Research Center.

Table of Contents