# *Kerberos With Clocks Adrift: History, Protocols, and Implementation*

Donald T. Davis and Daniel E. Geer
Openvision Technologies
Theodore Ts'o
Massachusetts Institute of Technology

ABSTRACT: We show that the Kerberos Authentication System can relax its requirement for synchronized clocks, with only a minor change which is consistent with the current protocol. Synchronization has been an important limitation of Kerberos; it imposes political costs and technical ones. Further, Kerberos' reliance on synchronization obstructs the secure initialization of clocks at bootstrap. Perhaps most important, this synchronization requirement limits Kerberos' utility in contexts where connectivity is often intermittent. Such environments are becoming more important as mobile computing becomes more common. Mobile hosts are particularly refractory to security measures, but our proposal gracefully extends Kerberos even to mobile users, making it easier to secure the rest of a network that includes mobile hosts. An advantage of our proposal is that we do not change the Kerberos protocol *per se*; by reinterpreting an unused challenge-response handshake in the standard Kerberos protocol, we convey just enough replay protection to authenticate the initial ticket and its timestamp to an unsynchronized client, without adding process-state to the system's servers. We have implemented this protocol in the MIT Kerberos V5 source-distribution.

 

I used to be Snow White, but I drifted.
—Mae West

## 1. Introduction

The Kerberos Authentication System [Steiner et al. 1988; Neuman & Ts'o 1994] provides password security for large networks. Unlike its principal competitors, KryptoKnight [Molva et al. 1992] and SESAME, [Parker 1991] Kerberos requires that all of a network's system clocks must be synchronized. At first glance, this does not seem to be a great burden, at least for UNIX networks, but as Kerberos' influence has grown, synchronization has become a substantial impediment to Kerberos' adoption as a uniform networking standard.

Why has clock-synchronization become more difficult? We find that for three areas of explosive growth in the networking industry, there are good reasons for rejecting clock-synchronization. First, in-house wide-area networks have only recently become common. Corporate wide-area networks usually arise by agglomeration, so interdepartmental rivalries often obstruct centralized host management, including time-synchronization. Such practical political strains were less important in the more monolithic academic and engineering networks that adopted Kerberos early. It's clear, though, that monolithic networks are now passé, so Kerberos will have to accommodate such ordinary social tensions if its success is to continue.

Second, online-access providers are bringing a massive surge of decentralized participants into the network industry. Obviously, it's commercially and technically infeasible to force synchronization on in-home network customers. Similarly, electronic commerce is bringing together buyers and sellers who share neither administrative nor organizational links; this openness guarantees that asynchronous clocks will remain the norm.

Third, the rise of mobile computing is bringing the problem of intermittent connectivity into renewed importance. Here, the problem is mainly technical: on slow connections, a time-synchronization protocol would burden both the laptop processor and its connection-initiation bandwidth.[1] Also, a social obstacle to synchronization is that the intermittent user may alternate among several organizations' networks; synchronization would force such users' clocks to flutter. Even

1. Eventually, faster network infrastructure will make extra exchanges transparent to the user, removing this obstacle to mobile synchronization.

as mobile users are more vulnerable to security breaches than sequestered networks are, their intermittency and mobility obstruct the most effective approaches to open systems security:

- Intermittency obstructs time-synchronized cryptographic protocols.
- At the same time, challenge-response protocols entail extra messages (see below), and these would impose unacceptable long-haul network delays on mobile users.
- Absent cryptography, the only alternative is firewalls and other protocol filters, but to a firewall, mobile users look like intruders.

This dilemma makes a synchronization waiver for Kerberos even more valuable. Once mobile users can authenticate themselves cryptographically, it becomes possible for a firewall or filter to recognize them, so that the home network can enjoy both styles of protection, instead of being denied both.

## 2. *Why Synchronize?*

In this section, we explain synchronization's purpose and alternatives that serve the same purpose, so as to review the history of synchronization's role in Kerberos' development.

Why does Kerberos need time synchronization in the first place? Synchronized clocks enable Kerberized applications to reject replay attacks. In a replay attack, an attacker eavesdrops on users as they present their credentials to servers; later, he resends the credentials to impersonate the users. A Kerberos client blocks replay by embedding an encrypted timestamp in each credential; the application server rejects credentials bearing out-of-date timestamps. Timestamps from users with slow clocks are indistinguishable from replays, so tolerating slow clocks gives attackers more time in which to work. Synchronization sharply limits this "replay window."

The alternatives to timestamping are all variations on "challenge and response." [Gong 1993] In a challenge-response protocol, the credential recipient prevents replay by challenging each sender to encrypt and return a fresh random number, so as to demonstrate timeliness. The sender proves his identity by using his private key, or his session key, to encrypt the random number. Challenge-response protocols avoid the complication of synchronizing, but they always use at least one more message than a timestamp protocol, to accomplish the same security goal. Thus, it might seem that Kerberos' designers chose to optimize performance with timestamps and synchronization. As it happens, though, this

speed/complexity tradeoff was *not* the reason Kerberos' designers chose a synchronizing protocol.

The 1978 Needham-Schroeder protocol, [Needham & Schroeder 1978] from which Kerberos descends, used challenge and response to protect authentication credentials from replay. Three years later, Denning and Sacco [Denning & Sacco 1981; Burrows et al. 1989] pointed out that the N-S protocol was particularly vulnerable to compromised session-keys, because its key-distribution tickets made no provision for expiration of keys. They recommended that the tickets be timestamped, so that the session-keys would expire and be renewed regularly. They also recommended replacing challenge-response with timestamps in N-S' session-authentication handshake, and they pointed out that minute-resolution clock-synchronization would suffice to enforce key expirations. Here, synchronization helps to ensure that every connection gets a new session-key. This precaution makes it less profitable to steal session-keys or to attempt their cryptanalysis. Unfortunately, Denning and Sacco did not discuss the importance and difficulty of securing the time-synchronization process itself.

In the mid-80's, MIT's Project Athena incorporated Denning and Sacco's recommendations into their implementation of the Needham-Schroeder protocol, and added other protocols and security features, too. [Steiner et al. 1988]. With timestamping in place, N-S became Kerberos' flagship protocol, which we at Athena christened the "Authentication Service." This protocol handles all of Kerberos' password-mediated authentication, principally initial logins and password changes. Kerberos' other protocols enable a logged-in user to authenticate to additional services without entering a password anew, and without retaining the password on the local machine.

These newer protocols uniformly use encrypted timestamps to block replay, following Denning and Sacco's recommendation. However, Kerberos was designed to accommodate clock-skews of up to five minutes between clients and servers (though modern time services can synchronize much better than this). Thus, a replayed authentication-message will not be rejected as out-of-date, if it's less than five minutes old (a generous allowance, though not an unreasonable one). To close this security hole, Kerberos introduced a "replay cache," in which an application server stores each encrypted timestamp it receives for five minutes, the duration of the replay window. Each server should check every new timestamp it receives against its cache, so as to block replays of "fresh" timestamps.

In 1990, 12 years after Needham and Schroeder's paper, and five years after Kerberos' introduction, Bellovin and Merritt of AT&T Bell Labs wrote an important and insightful critique of Kerberos' version 4, which was influential in the design of the current version 5 [Bellovin & Merritt 1990]. Along with other

problems, Bellovin and Merritt pointed out that Kerberos security depends on *secure* clock-synchronization, and that V4 Kerberos was not itself sufficient to secure a clock-synchronization service. The clearest demonstration of this insufficiency is to consider a computer that is restarting automatically from a power failure, so that its system clock is probably unreliable. In this situation, the computer cannot be sure of any message's freshness; indeed, if an attacker replays all of a previous day's network traffic, he can mislead the computer into using an old, compromised session-key as if it were fresh, and Kerberos' guarantees evaporate. As Bellovin and Merritt noted, the only way to defeat such an attack is with a challenge-response protocol, which Kerberos currently lacks, and which no current time-service supported.

It turns out that this situation is not merely illustrative, but is actually the crux of the problem. Only when a Kerberos principal first comes onto the net, does he need to use a challenge-response handshake to prevent credentials-replay. However, application clients and servers enter the network differently, so they must handle synchronization differently, too. Application servers need to use a challenge-response handshake only at bootstrap, to get time-service tickets. Thereafter, a server can trust its system clock, whenever it needs to renew its time-service tickets or other tickets it uses. For application clients, challenge-response is necessary whenever the user logs on to a physically-insecure workstation. Once the challenge-response handshake has assured the client of his initial tickets' freshness, the client does not need to synchronize his clock with the rest of the network. To be able to detect replay, the client only needs to know the difference, or skew, between his clock and the standard clock [Zanarotti 1995]. Thus, by adding a challenge-response handshake to only the Authentication Service protocol, we can break the circularity of Kerberos' dependence on a secure time-service.

## 3. *Current Time Services*

NTP is a cryptographically-hardened time service protocol [Mills 1989.12; Mills 1989.13]. It enables a wide-area network to synchronize its software clocks with a few highly-accurate physical clocks. NTP's security has been extensively analyzed by Matt Bishop [Bishop 1990]. Each secure clock update depends on an uninterrupted chain of authentications, server-to-server, between the client and a remote physical clock. To mediate these authentications, NTP requires each host to maintain a shared key in a disk file, but makes no provision to distribute or refresh these keys. Kerberos can manage NTP's keys, but only under the assump-

tion that the clocks are already synchronized. NTP makes no claim to solve this bootstrap problem; it assumes that secure key-management is available as reliable infrastructure, just as Kerberos assumes that secure time-synchronization is available. Despite its reliance on out-of-band key-distribution, secure NTP is a suitable mechanism for Kerberos servers to synchronize their own clocks, because every Kerberos server has to share a key with each of its peer servers, anyway. We do not recommend using secure NTP for synchronizing application services, however.

The Open Software Foundation's Distributed Computing Environment (OSF DCE) includes a secure Distributed Time Service [Open Software Foundation 1992], whose security is mediated by DCE's Kerberos-based Security Service. For bootstrap, the DCE time service relies on the host's hardware clock chip to be physically secure, battery-powered, and accurate enough to fulfill Kerberos' secure synchronization needs. DCE explicitly accepts, just as Kerberos always has, that the clocks must be initialized "out-of-band," i.e., by wristwatch [Pato 1995]. DCE's DTS is designed to interoperate with NTP, but this interoperation does not address our bootstrap problem. Finally, neither NTP nor DCE's DTS makes any provision for physically-insecure hosts, which cannot hold long-lived keys on disk, and which therefore cannot participate in either protocol. Our proposal will work well with both of these services, without substantial change to their protocols or software.

## *4. Secure Synchronization*

In this section, we describe a challenge/response protocol for Kerberos, that enables users to get tickets without having synchronized their clocks. This C/R protocol is already present in the Kerberos V5 specification. It was intended to implement part of Bellovin and Merritt's suggestion that Kerberos abandon timestamping altogether, which was their solution for Kerberos' circular reliance on secure time service. Our use of C/R differs from Bellovin's recommendation in four ways:

1. The Kerberos client uses the C/R protocol only once, upon connecting to the network at login or bootstrap.
2. After connecting, the client uses and checks timestamps to prevent replay.
3. Our Kerberized application servers never use C/R to prevent replay at all.
4. Our proposal adds no process-state to the Kerberos server or to the application servers.

Thus, our contribution is to observe that a single, standard C/R exchange suffices to break the circular dependence between Kerberos and secure synchronization.[2]

For clarity's sake, let's consider first how to initialize a clock securely, on a machine that does intend to synchronize. Suppose Bob is a system server who shares a key $K_b$ with the Kerberos Authentication Server $AS$, and suppose he is willing to synchronize his clock. Then to do this, every time he reboots, one of his tasks will be to get tickets from the Kerberos Server, or KDC.[3] For now, it doesn't matter what service's tickets Bob requests, but call the service $S_t$. Bob will send a nonce $N_b$ in a challenge-response handshake:

$$B \rightarrow AS: \ B, \ S_t, \ N_b \tag{1}$$

$$AS \rightarrow B: \ T_{bt}, \ \{S_t, \ N_b, \ L, \ K_{bt}\}^{K_b} \tag{2}$$

Bob's nonce $N_b$ is a random number, which he can generate from disk-drive randomness [Davis et al. 1994] or from some other noise source [Eastlake et al. 1994]. The *AS* returns to Bob a new session-key $K_{bt}$, the key's times of creation and expiration $L = (L_{create}, \ L_{expire})$, a ticket $T_{bt} = \{S_t, \ L, \ K_{bt}\}^{K_t}$, and the nonce $N_b$, newly encrypted. Except for the nonce contents, Bob's exchange is identical to a usual Kerberos initial ticket request. By changing the contents and their interpretation, we've strengthened the ineffective challenge-response handshake that's already in the standard login protocol.

This handshake proves to Bob that the key $K_{bt}$ and ticket $T_{bt}$ are fresh. On receiving his tickets from $AS$, Bob decrypts them with his password $K_b$. When Bob finds that his password-key $K_b$ was indeed used to encrypt his nonce $N_b$ and his new session-key $K_{bt}$, he concludes that $AS$ prepared the tickets after receiving $N_b$. As long as he has never used $N_b$ to request tickets before, this means that the tickets are fresh. At this point, Bob can use $L_{create}$ to reset his clock. It is important that Bob's choice for $N_b$ must be immune to external influence; if an attacker can cause Bob to re-issue an old challenge $N_{old}$, then she can replay correspondingly old credentials $T_{old}$, $\{S_t, \ N_{old}, \ L_{old}, \ K_{old}\}^{K_b}$, whose session key $K_{old}$ she knows by prior theft.

2. In an earlier version of this article [Davis & Geer 1995], a similar C/R protocol was presented which exploited Kerberos V5's flexible preauthentication feature [Pato 1992; Neuman & Kohl 1993] for secure synchronization. While we were designing the implementation of this earlier, preauthentication-based scheme, Cliff Neuman pointed out that Kerberos Version 5 KDC_AS_REQ and KDC_AP_REQ messages already contain nonce fields, and that these fields are already being used to provide C/R authentication of the KDC to the client. However, the Kerberos protocol specification [Neuman & Kohl 1993] warns against using the *authtime* field in the KDC_AP_REP message for time synchronization. This warning indicates that this nonce was originally intended *not* to support synchronization, but merely to provide mutual authentication of the KDC to the client.
3. "Key Distribution Center" is a generic term for the Kerberos Server's role.

As in the usual Kerberos protocol, *AS* learns nothing from this exchange about whether it really was Bob who requested tickets, unless he uses preauthentication data to authenticate his request [Pato 1992; Neuman & Kohl 1993] Note though that when preauthentication is available, it may make our challenge-response unnecessary. Many preauthentication mechanisms, such as smart-card protocols, were originally designed as mutual-authentication schemes in their own right, and do authenticate the server to the client. As long as the preauthentication protocol also protects the initial Kerberos credentials from replay, the client can trust the creation-time to represent Kerberos' current clock-time, without having to use our challenge/response handshake.

With the above protocol, the server Bob only synchronizes once, but eventually his clock will drift away from the KDC's clock again. To maintain his synchronization accurately, Bob can use the C/R protocol to request Kerberos tickets for a secure time-service. As above, his ticket's creation-time will get his session-clock into close agreement with the KDC's version of Universal Time[4], so that he can then use his time-service credentials to keep his clock exactly synchronized with the time-service. When Bob uses his new time-service ticket, he sends the usual authenticator, or encrypted timestamp.

$$B \rightarrow S_t : \ T_{bt}, \ \{B, \ time\}^{K_{bt}} \tag{3}$$

Note that since Bob has recently set his system clock from the ticket's creation time, his *time* will be fairly accurate. Because Bob requested mutual authentication, the time server returns the timestamp *time* from Bob's authenticator. In addition, the time server returns the correct time t.o.d.:

$$S_t \rightarrow B : \{time, \text{“UT} = t.o.d.\text{”}\}^{\mathrm{K_{bt}}} \tag{4}$$

The first part of the server's response assures Bob that the time-report is fresh, because it echoes the timestamp he sent. Recall that Bob knows that his timestamp was fresh, because he got it from the KDC's C/R-authenticated tickets.

For a user Alice, unlike the application server Bob, secure synchronization is much easier, because she doesn't need to synchronize for long periods. All Alice has to do is run an *insecure* time-service, and check it against Kerberos' authenticated time, whenever she logs in. As long as the two servers' times are nearly equal, Alice can safely trust the time-server's management of her system-clock. If a Kerberos site chooses not to run secure NTP, GPS, or some other secure time-service for its KDC, the Kerberos server itself can use Alice's trick to validate

4. By Universal Time, we mean the time as measured by reference clocks which are maintained by national and international organizations.

NTP updates from outside its realm. Each KDC can get AS tickets from another realm's KDC; this secure time-source is not exact, but is close enough for sanity-checking the exact but insecure time-service.

## 5. *Nonce Preparation*

The most difficult part of this proposal is the client's task of preparing a unique nonce. Traditionally, nonces are thought of as cryptographically-random numbers, and the lack of true randomness has long been an obstacle to the implementation of challenge-response protocols in software security systems. Strictly speaking, though, our nonce does not even have to be unpredictable, much less random. In fact, it suffices for the client to use an incremented counter for his challenge, as long as the user can be sure not to use the same value twice from the same workstation. In this section, we will describe both ways to create nonces, and we'll discuss their merits and limitations.

A predictable nonce is much the easier solution. The client workstation just keeps a counter in a file that users cannot alter, and uses this counter for nonces. For good security, it would be unwise therefore to use this technique on a public-access workstation, on any remote-login server, or on any machine that isn't physically secure. A user does not have to worry about re-using the same nonce value on different machines, though, because the Kerberos protocol will recognize the difference in encrypted IP addresses.

A random nonce is more versatile, but is more expensive in two ways. True natural randomness is available on almost every computer, because the computer's disk-speed varies slightly but constantly. This speed variation originates in turbulent air-drag on the disk platters, and is big enough to subtly perturb the timing of disk-accesses [Davis et al. 1994]. The cost of disk-randomness is its low bandwidth and its dependence on the operating system, CPU, disk-drive, and controller board. It is possible to avoid these dependencies while gathering the timing-data, but the dependencies still strongly influence the rate at which measurable entropy can be extracted. Depending on the platform, anywhere from 1 bit per minute to 20 or more bits per second may be available from disk-noise.

Another readily-available source of natural randomness is keyboard-timing [Eastlake et al. 1994], but this is problematic, too. First, when the Kerberos client software is preparing its first ticket-request, the only keyboard-input it sees is the user's entry of his username; the password doesn't get read until after the KDC reply is received. So, very little noise is available from this source. Second, the Kerberos routines currently use device-independent string I/O routines to read

these strings; equipping Kerberos to time the keystrokes would, again, require a lot of device-dependent code.

The only other sources of variability for nonce-generation are the username, machine name, process-ID, and time of day. By themselves, these are not random, but if they are hashed together with whatever natural randomness can cheaply be gathered, they'll help make a unique nonce for the session. Because I/O timing can influence the shuffling of the Kernel's data-structures, Kernel-memory contains a limited record of some past I/O noise, so a hash of `/dev/kmem` might be useful, too.

It's important to note that the practical utility of our synchronization proposal relies entirely on a satisfactory source of cryptographic randomness for Kerberos' use. Ideally, all computers would be equipped with hardware random number generation devices. This would provide high-quality random numbers from a quantum source (e.g., a noise diode). In addition, operating systems should provide a standardized interface for obtaining high-quality random numbers. [5] Unfortunately, such hardware and operating-system support is not common today, though we hope it will become a standard part of every computer. Until this support arrives, the practical utility of our synchronization proposal will have to rely on a special-purpose implementation of disk-randomness for Kerberos' use.

## 6. *Session Clocks*

In this section, we describe a mechanism that enables users to present accurate timestamps to Kerberos and to secure applications, without keeping their system clocks synchronized. Suppose now a user Alice wishes to communicate securely with Bob, but suppose that like most users, she prefers *not* to synchronize her clock.

In this case, Alice won't request time-service tickets, but she still needs to keep track of the Kerberos server's clock-value, so that she can prepare acceptable credentials, detect replays herself, and anticipate her tickets' expiration. Her ticket's lifetime data $L$ tell her the current value of Kerberos' clock, because $L$ includes the ticket's creation-time $L_{create}$. Alice can record the skew $\Delta_a = L_{create} - time_a$ between her clock and Kerberos', so as to keep track of

5. For example, Theodore Ts'o has developed a `/dev/random` I/O driver for Linux. This driver provides high quality random numbers by sampling environmental noise from events such as keyboard interrupts and disk-timing. In practice, operating system kernels tend to be able to collect environmental noise much more efficiently than user-mode programs.

Kerberos' clock's value. This fixed skew will enable her to prepare acceptable credentials, etc. as usual. [6]

Alice begins her login-session by asking $AS$ for a ticket-granting ticket (TGT), which she'll then use to request tickets for Bob's service:

$$A \rightarrow AS : A,\ TGS,\ N_a \tag{5}$$

$$AS \rightarrow A : \{TGS,\ N_a,\ L,\ K_{a,tgs}\}^{K_a},\ T_{a,tgs} \tag{6}$$

This is the same challenge-response handshake that Bob used above, except for the names. On receipt, Alice concludes that her ticket and session-key are fresh, just as Bob did, and she uses the key's creation-time $L_{create}$ to construct a normal TGS ticket-request:

$$A \rightarrow TGS :\ B,\ T_{a,tgs},\ \{A,\ L_{create}\}^{K_{a,tgs}} \tag{7}$$

$$TGS \rightarrow A :\ T_{ab},\ \{B,\ L',\ K_{ab}\}^{K_{a,tgs}} \tag{8}$$

Now, to detect replayed TGS-replies, Alice can compare her new ticket's creation-time $L'$ with $time_a + \Delta_a$, which will be a good approximation to $time_{AS}$.

Note that after her initial login with challenge and response, Alice's other security interactions are perfectly standard, and the rest of the Kerberos protocol is unchanged. However, to support drifting-clock clients (those who avoid synchronizing), the Kerberos application library would have to be changed to maintain transparently an implicit "session clock" at each end of a Kerberized connection. Each side's skew $\Delta_{local} = time_{AS} - time_{\text{local}}$ would be initialized at login or at bootstrap; thereafter, whenever the Kerberos library needs synchronized time, it would add the skew to the local clock. This would allow an application's client and server to use Kerberos for security, even though neither party has synchronized his clock with Kerberos. Similarly, when a client interacts with several Kerberos servers, he'll have to maintain a separate clock-skew for each one (unless all KDCs synchronize with Universal Time, as we recommend).

Because clocks drift, Alice's calculated value for $\Delta_a$ will grow stale eventually, perhaps after several days. This drift is too slow to affect human users, who must daily refresh their tickets (and thus their clock-skews) anyway. But if a server Bob wishes to maintain a clock-skew $\Delta_b$, he must interact daily with Kerberos to refresh $\Delta_b$ from his tickets' $L_{create}$.

Because the Kerberos protocol is unchanged, the session-clock clients and synchronized clients would be indistinguishable in their network behavior.

6. Stan Zanarotti, of Dimensional Insight, Inc., devised an unsecured version of this clock-skew trick at MIT, when he implemented MIT's Kerberos clients for the Apple Macintosh. The trick is particularly necessary for the Mac, whose clock is hard to keep synchronized for a variety of reasons [Zanarotti 1995].

Session-clock Kerberos clients and servers would fully interoperate with a normal Kerberos installation, because the Kerberos server already handles the nonce-fields correctly, and does not have to be changed. Only the client-side code changes. The only change in its behavior is that at login, the client sends and checks a real nonce, instead of a timestamp, and he constructs his subsequent timestamps in a novel way. These differences are invisible to the protocol, and can't cause a version-skew.

## 7. *Closing the Replay Window*

Some applications use Kerberos only to authenticate the client and server at the beginning of their sessions. When such a server crashes, and until it restarts, it is unable to maintain its replay-cache. Thus, if a client sends a timestamped service-request during the downtime, an attacker can replay the request after the service restarts, without fear of detection. Clearly, the application can minimize this threat by integrity-checking the request, so that the attacker can't alter it. Even integrity-checking, though, leaves the attacker with the ability to change the request's timing, within the constraint of the timestamp-lifetime. For example, in a securities-trading application, an attacker might profit simply by replaying someone else's bounced trading request, just after a server restarts.

This restart replay-window is only a minor security hole, but it's worth closing, especially for applications that don't enforce integrity-checking. We know of only two ways to protect absolutely against replays of downtime requests:

1. Wait for the downtime requests to expire: the server should reject any timestamps that predate its restart, on the assumption that they may be replays.
2. Enforce integrity-checking in a single-server two-phase commit protocol, so that the attacker can't forge the "commit" verification.[7]

Unhappily, neither of these alternatives can apply universally. On the one hand, some applications, such as UNIX's `inetd`-mediated services, start afresh for each request, so they cannot know when they were started. On the other hand, some services can't or won't support the complexities of integrity-checking and two-phase commit protocols.

It seems that the only prudent alternative is to shorten the timestamp lifetime, so as to narrow the replay-window. This doesn't completely block the "downtime

7. This also prevents "suppress and replay" attacks.

replay" attack, but it makes the attack harder to perform. Thus, one advantage of building time-synchronization directly into the Kerberos protocol and its libraries is that once we can take synchronization for granted, we can narrow the five-minute replay window.

MIT's Kerberos developers chose five minutes as Kerberos' timestamp lifetime somewhat arbitrarily. At that time, in the mid 1980's, time daemons weren't yet widely available, and network latencies were bigger than they are now. So, five minutes seemed a good trade-off: a shorter lifetime would be hard to manage with manual synchronizations, but a longer lifetime seemed to ask for trouble, given that we hadn't built the replay-caching code yet.

How much can we reduce the replay-window? We still have to allow for network latency. In fact, the ages of our session-clock timestamps always appear at their receipt to be roughly twice the network-latency. This means that the timestamp-lifetime must exceed twice the usual maximum for the network's latency. At this writing, a Boston-to-Auckland round trip runs about 250 milliseconds, so a lifetime of 5 to 15 seconds seems adequately conservative.

More precisely, if $\Delta_{k,c}$ is the latency between the client and the KDC, and if $\Delta_{c,s}$ is the latency between the client and the application server $S$, then when the client sends a timestamp to the application server, the timestamp will lag Universal Time by $\Delta_{k,c} + \Delta_{c,s}$. If the server is using a session-clock, too, then the server's session-clock will lag UT by $\Delta_{k,s}$, and the timestamp's age will seem to the server to be

$$\Delta_{k,c} + \Delta_{c,s} - \Delta_{k,s}.$$

On average, we can expect this apparent age to be non-negative, because the direct path between the KDC and $S$ should be shorter than the relayed path via $C$. (Of course, if $S$ synchronizes with UT, then the apparent age will always be strictly positive.) Thus, in either case, an honest client's session-clock will not prepare a postdated timestamp. This blocks a "postdated-timestamp" attack noted by Li Gong [Gong 1992]. Gong points out that if a client's synchronization fails so that his clock runs fast, his accidentally-postdated message can be suppressed and replayed later, when the postdated message becomes current. In this way, an attacker can gain an advantage from a time-sensitive service, even after the client corrects his clock.

## *8. Implementation*

Our implementation of this clock synchronization scheme was quite straightforward. The Kerberos V5 protocol already contained a nonce field in the

KRB_AS_REQ message, which the Kerberos server is required to copy into the encrypted portion of the KRB_AS_REP message. Hence, no server-side changes were required. This provided us with the happy result that all of the existing, installed Kerberos servers required no modifications in order to support clients wishing to use this new time synchronization technique.

The changes to client code were relatively simple. The `krb5_get_in_tkt()` routine already passed a nonce to the Kerberos server, and verified that Kerberos server's reply contained the same nonce. All that we needed to add was a few lines of code to sample the client's system clock, and then compute an offset between the system time and the value of the *authtime* field from the KDC_AS_REP message.

We then changed the `krb5_gettimeofday()` function to reconstruct the Kerberos server's time from this offset and the local system's time. This routine is called by all Kerberos client routines when they need the current time for Kerberos protocol messages.

For security reasons — the same reasons why unprivileged users must not be allowed to change the system time — this time offset must be stored separately for each user. We store the user's session-clock offset in the header of his credentials cache (i.e., the ticket file, in Kerberos V4 parlance). Although storing the clock offset in the credentials cache is somewhat impure from an architectural point of view, from a practical point of view it works very well. The credentials cache is generally reinitialized when the user obtains a fresh TGT using the Kerberos Authentication Server protocol, and the cache is referenced each time the client authenticates itself to an application service. We create and use the offset at these same times, so it's natural and convenient to store the offset in the credential cache.

The only place where some care is required in the client implementation is in generating a nonce. If Alice reuses a nonce to gain tickets, and if the session key from the nonce's first use has since been compromised, an attacker can successfully impersonate the KDC by sending the old, compromised tickets to Alice. Alice will also receive the reused nonce, correctly encrypted, and so she will accept the old, compromised ticket and session-key as fresh. This ticket is outdated, and so would not be accepted by any legitimate application server. However, the stolen session-key enables the attacker to impersonate the application server, too. Thus, when the attacker knows an old session-key and Alice re-issues the corresponding nonce, she can be tricked into thinking that she had mutually authenticated with an application server, when in fact she was sending confidential data to the attacker instead.

MIT's current V5 beta 5 implementation used the system clock to generate a nonce, as recommended by the Kerberos protocol specification. However, this

assumes that the system clock is accurate, so that the value of the system clock is always increasing. Since we are trying to use the Kerberos protocol to synchronize the system clock, we cannot make this assumption.

Since the nonce does not need to be unpredictable, but only non-repeatable, one possible solution is to store the previous nonce, and then increment it to obtain the next value to be used. This solution requires that the machine be able to store the nonce reliably and securely. Unfortunately, while storing process-state can be workable for application servers, it's often infeasible for clients to store process-state securely.

Lacking a system clock which is always increasing and the ability to store state, the remaining solution is to use a randomly picked nonce value. However, generating truly random and unpredictable values can be very difficult, since by and large, computers are carefully engineered to generate *repeatable* results. Eastlake, Crocker, and Schiller have pointed out the practical difficulties in generating truly random values for cryptographic purposes [Eastlake et al. 1994].

In our "proof of concept" implementation, we adopted the strategy of storing the previously used nonce in a file, primarily for ease of implementation. Our implementation can be extended to allow this time synchronization technique to be used on clients which cannot reliably store state. However, this would require that that the implementation generate and use truly random nonces. We invite the interested reader to look to [Eastlake et al. 1994] for suggested implementation hints. Unfortunately, many of these techniques either require user input, which is not always available to the Kerberos library, or are extremely dependent on hardware and operating system configurations.

We make the assumption that the Kerberos servers are running with their system clocks set to the correct time. This is important, since under this scheme, clients and servers synchronize their clocks only with their local Kerberos server. In order for clients and servers in different realms to be synchronized (which is required for inter-realm authentication), the system clocks of all of the Kerberos servers must be in sync. We therefore strongly advise that all Kerberos servers run some sort of network time-synchronization protocol, such as secure NTP, or failing this, use an external device, like a WWWV receiver or a GPS receiver, in order to maintain the Kerberos server's clock. These alternatives assure that Kerberos servers and their clients will securely be synchronized with Universal Time.[8]

8. Where secure and exact synchronization isn't possible, a Kerberos server can use the C/R protocol with another KDC to validate an insecure NTP service, just as a client can do (see Section 4).

## 9. Conclusion

We have presented a solution to Kerberos' "cold-start" problem in clock synchronization, which provides for secure clock initialization where needed, and for "drifting clock" security where desired. We expect the proposal to gain acceptance rapidly in the broad community of Kerberos' vendors, implementors, and designers, because it requires only minor changes to the Kerberos client library and to the secure time protocols, and because it adds no extra network delays to users' login sequence. Indeed, for Kerberos implementations that already employ preauthentication to protect against dictionary attacks, our proposal requires little more than a shift in interpretation, to exploit the fact that with some preauthentication schemes, V5 beta 5 Kerberos tickets already can be trusted to deliver a secure clock-initialization.

We also expect our protocol and its implementation to greatly improve Kerberos' attractiveness to a variety of commercial network customers and users. Our notion of relaxed, yet secure, synchronization will further lighten administrative burdens and enhance security in large networks. It actually reduces Kerberos' administrative overhead, since most client machines will be able to dispense with time daemons, and it adds neither overhead nor network-latency to secure applications.

Intermittency, more than anything else, is the core technical challenge of mobile computing, yet mobile, intermittently connected counterparties have a bigger stake in authenticity than do continuously connected, sequestered network environments. As such, we claim that easing Kerberos' synchronized clocks constraint is uniquely valuable, because it enables the efficiency and prompt, assured revocation of authority (that is the hallmark of Kerberos authentication) to be broadly applicable to environments that do not and will not have time-synchronization services. More broadly, we suggest that as the demands of electronic commerce become better understood, the ability to bridge the boundaries of internally synchronized yet mutually unsynchronized organizations will be shown to have compelling value.

## 10. Acknowledgments

We thank Barry Jaspan, Marc Horowitz, Jonathan Kamens, John Kohl, Cliff Neuman, Joe Pato, and Stan Zanarotti for their helpful comments and suggestions.

## References


1. S. M. Bellovin and M. Merritt, Limitations of the Kerberos Authentication System, in *USENIX Conference Proceedings*, pages 253–267 (Dallas, TX; Winter 1991). Also in *ACM Comp. Comm. Rev.*, **20**(5), pages 119–132 (October 1990), `research.att.com:dist/internet_security/kerblimit.usenix.ps`.
2. M. Bishop, A Security Analysis of the *NTP* Protocol, *Sixth Annual Computer Security Conference Proceedings*, pages 20–29 (Dec. 1990; Tuscon, AZ), `louie.udel.edu:/pub/ntp/doc/security.ps.Z`.
3. Michael Burrows, Martín Abadi, and Roger Needham, A Logic of Authentication, *Proc. R. Soc. Lond. A* **426**(1989), pages 233-271.
4. D. Davis and D. Geer, Kerberos Security with Clocks Adrift, *Proc. 5$^{th}$ USENIX UNIX Security Symposium*, Salt Lake City (June 1995), pages 35–40.
5. D. Davis, P. R. Fenstermacher, and R. Ihaka, Cryptographic Randomness from Air Turbulence in Disk Drives, in *Advances in Cryptology—CRYPTO '94*, Ed. by Yvo G. Desmedt. Springer-Verlag Lecture Notes in Comp. Sci. **839**, pages 114–120 (1994).
6. D. Denning and G. M. Sacco, Timestamps in Key Distribution Protocols, *CACM* **24**(8), pages 533–536 (August 1981).
7. D. Eastlake, S. Crocker, and J. Schiller, *Randomness Recommendations for Security*, Internet RFC 1750, December 1994.
8. L. Gong, A Security Risk of Depending on Synchronized Clocks, *ACM Op. Sys. Rev.*, **26**(1), pages 49–53 (1992), `http://www.csl.sri.com/gong/pub393.html`.
9. L. Gong, Variations on the Themes of Message Freshness and Replay, *Proc. IEEE Computer Security Foundations Workshop VI*, Franconia, NH (June '93), pages 131–6, `http://www.csl.sri.com/gong/pub393.html`.
10. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Project Athena Technical Plan*, Sec. E.2.1: Kerberos Authentication and Authorization System, (Cambridge, Mass.) M.I.T. Project Athena internal document, Dec. 21, 1987.
11. D. L. Mills, *Network Time Protocol (Version 2) Specification and Implementation*, Internet RFC 1119 (Sept. 1989).
12. D. L. Mills, *Internet Time Synchronization: the Network Time Protocol*, Internet RFC 1129 (Oct. 1989).
13. R. Molva, G. Tsudik, E. van Herreweghen, and S. Zatti, KryptoKnight Authentication and Key Distribution System, *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, Toulouse, France, November 1992. `jerico.usc.edu:pub/gene/kryptoknight.ps.Z`.
14. R. M. Needham and M. D. Schroeder, Using Encryption for Authentication in Large Networks of Computers, *CACM*, **21**(12), pages 993–999 (December, 1978).
15. C. Neuman and J. Kohl, *The Kerberos Network Authentication Service (V5)*, Internet RFC 1510, September 1993.

16. C. Neuman and T. Ts'o, Kerberos: An Authentication Service for Computer Networks, *IEEE Communications*, **32**(9), pages 33–38 (September, 1994).
17. Open Software Foundation, *$OSF^{TM}$ DCE Version 1.0, DCE Administration Guide*, Volume 1, Module 4: DCE Distributed Time Service, Rev. 1.0, Update 1.0.1. (Cambridge, MA; July 1992).
18. T. A. Parker, A Secure European System for Applications in a Multi-Vendor Environment (The SESAME Project), *Proc.* $14^{th}$ *Am. Nat'l. Sec. Conf.*, 1991.
19. J. Pato, *Using Pre-Authentication to Avoid Password Guessing Attacks*, (Cambridge, Mass.) M.I.T. Project Athena (December 1992).
20. J. Pato, personal communication, 1995.
21. J. G. Steiner, C. Neuman, and J. I. Schiller, Kerberos: An Authentication Service for Open Network Systems, *USENIX Winter Conference Proceedings*, February 1988. `athena-dist.mit.edu:pub/kerberos/doc/usenix.PS`.
22. S. Zanarotti, of Dimensional Dynamics, Inc., was the first to use this trick; personal communication, 1995.