Global Information Assurance Certification Paper

Securing Time – The Autokey Protocols

R.H. Palko

Table Of Contents

1. Preface:

This paper investigates the authentication protocols used with NTP-V4. It does not review the NTP protocol itself, nor does it cover in detail the authentication protocol used in its predecessor, NTP-V3.

For those new to the topic, NTP is the Network Time Protocol, used to acquire a reliable time standard for a site/host from the Internet. A brief overview of the full NTP protocol can be found in Addendum 1.

NTP Authentication is unique in that it must operate in an initial environment of un-trusted sources coupled with inaccurate clocks. The problem is exacerbated by computational overhead constraints which impact the ultimate accuracy of the timestamps required for proper operation of the NTP Protocol. These unique requirements are why standard techniques such as IPSEC, and the naïve approach of signing each timestamp message are inappropriate for use as an authentication mechanism.

2. Overview:

The NTP Authentication schemes are for the purpose of identifying the Server to the Client, (the converse of the SSH model), to ensure the Client 'knows' the source of its time reference.

NTP-V4 has added 'extension fields' to the NTP message to convey cryptographic information between machines. These fields are only present in messages transporting association-id and cryptographic related information and are dispensed with in ordinary timestamp messages.

NTP Autokey is a "Work in Progress" on the standards track. This paper discusses the Authentication portion of NTP-V4 as distributed in the current reference implementation tar ball, ntp-4.0.99m-rc2.tar.gz.

Since NTP is a 'work in progress' it is subject to change. Therefore, this paper may be rendered obsolete in some parts by future distributions of the program suite.

Autokey does not employ encryption per-se, only hashing and signing functions. It does this by using a combination of PKI and pseudo-random one-time keys generated by the MD5 hashes. The NTP timekeeping mechanism itself provides some defense against false timestamps inherent in its filtering algorithm, but assumes the sources of the false timestamps are misconfigured Servers rather than by malicious intent.

In the reference implementation, Autokey is based on the RSAREF20 cryptographic library, namely the RSA public-private key system, MD5 and the Diffie-Hellman key negotiation

system.

NTP has to meet two requirements when authenticating, obtaining accurate timestamps and authentication information. A Peer or Server is not accepted into the host's candidate population until both Authentication Values and Network Delay Values have been obtained. Authentication is via Keyed MACs, (Message Authentication Codes), and RSA signatures of cryptographic data. Network delay calculations are based on timestamps. Timestamps are accepted unauthenticated for calculation purposes, but not used for clock synchronization purposes until authentication is complete and conversely. Once both items have been completed, the Client accepts the Peer or Server as a candidate source of synchronization.

There are two schemes available in NTP-V4, symmetric key, (backwards compatible with NTP-V3), and Autokey which is new as of NTP-V4.

Symmetric key authentication, (introduced in NTP-V3 and not to be confused with the NTP-V4 Symmetric Modes), utilizes either DES-CBC or Keyed MD5. These keys must be distributed by a secure means outside the NTP protocol suite, which leads to the usual key distribution conundrum encountered with any symmetric key system, in that $\sum_{k=1}^{n-1} k$ sets of keys have to be distributed, where n is the number of correspondents. Nothing further will be discussed about the NTP-V3 scheme. Interested parties are referred to RFC 1305.

Autokey solves the key distribution problem through PKI techniques, and uses a triad of algorithms; RSA Public/Private Key, Keyed MD-5 and Diffie-Hellman key exchange in its operations.

General protocol usage is MD5 to detect message modification, RSA to verify sources, and Diffie-Hellman to generate a common secret value in certain modes.

There are three sets of Autokey protocols corresponding to the three primary modes of NTP: Client-Server, Symmetric Peer, and Multicast*.

RSA Signatures with timestamps are used in all modes to verify the source of all cryptographic identities.

Session keys, extracted from a key list, are used in all modes for MD5 hash keys.

The Session key is a one-time key. For Key generation details see Addendum 2.

The NTP message authenticator is composed of two fields, the 'key/algorithm identifier' field and the 'mac digest' field. In authenticated NTP-V4 the 'key/algorithm identifier' field contains the K_{id}, (hereinafter referred to as the 'Auth.K_{id}') of the current $K_{session}$ key and the 'mac-digest' field, (hereinafter referred to as the 'Auth.MAC'), is the $K_{session}$ keyed md5 hash of the message fields.

Cookies are used in all modes and can assume several values dependent on the NTP mode. See Addendum 3 for details.

Session key lists are refreshed approximately hourly based on the polling interval. DH parameters and RSA keys are refreshed, on average, once per day. NTP-V3 private keys are permanent keys and are refreshed manually.

RSA Public keys and DH Parameters are public values and can be obtained via the Autokey protocols, or from sources outside the NTP system using insecure means such as file transfer.

No state information is kept by the Server in Client-Server or Multicast Modes. However partner state is kept in Symmetric Mode between Peers.

The authentic bit in the NTP status word is set when either the Cookie value, or Autokey key list values, (dependent on NTP mode), and RSA signatures are valid. If the authentic bit is set, the timestamps are evaluated otherwise they are ignored.

If the Client clock is stepped**, rather than slewed**, all cryptographic and time values for all associations are purged and the Autokey protocol is restarted, this is to ensure that these values do not survive a clock reset.

The most recent timestamp for each signed parameter is saved. Extension fields with timestamps which are zero or older than the saved timestamp for the related signature parameter are discarded without further processing.

While host system resident RSA key files and DH parameter files are not signed, their file extensions contain 'time of generation' timestamps.

Since the RSA public key timestamp is copied from the file extension timestamp, the files should always be generated when the host clock is valid.

A specific relationship amongst timestamps is enforced by the protocol:

1. All signature timestamps must be earlier than the packet receipt timestamp.
2. All signature timestamps must be later than the Public Key timestamp.
3. In Multicast Mode, the cookie timestamp must be later than the 'Autokey-values' timestamp.
4. In Symmetric Modes, The 'Autokey-value' timestamp must be later than the DH public key timestamp.

One caveat in the foregoing relationship structure, the granularity of the public key and signature timestamps is one second, so a difference of zero is ambiguous. Additionally, timestamps can be in error by as much as the synchronization distance** =

(root-dispersion+(root-delay/2)).

2.1 Error Handling:

There are two mechanisms in the protocol to handle errors, the reachability register and the watchdog counter.

The reachability register is an 8 bit left-shift register, which shifts a zero in for every poll interval. When a response packet is received which passes authentication and sanity** checks, a one is shifted into the register. If the register goes to zero, a general reset on the Association is done which restarts the protocol from the value state obtaining when the Association was first created.

The watchdog timer is incremented at each poll interval, regardless of the state of the reachability register. The timer is cleared when a cryptographically authenticated packet is received. If the counter reaches a value of 16 a general reset is done on the Association. When the Association is restarted the poll interval is doubled.

The default NTP message error recovery mechanism is that a request is repeated at each poll interval until a response is received or the watchdog timer expires causing a general reset.

2.2 Security Issues:

One possible attack mode is a denial of service attack by forcing the Client to expend cycles running expensive key refresh routines by sending bogus or replayed signatures.

To defend against this attack, all signatures are time stamped. The timestamp is either a valid NTP-seconds stamp or has the value of zero if the Server is not synchronized. Extension fields with a timestamp value of zero are discarded immediately without further processing.

The cryptographic values are smaller than those normally associated with strong cryptography. This is driven by two constraints, the requirement for backward compatibility with NTP-V3 and the need to minimize the processing overhead associated with cryptographic calculations.

There are three tiers of security defense:

1. Message Authentication Code, (MAC):
   Any message received with an invalid Auth.MAC field is discarded immediately without further processing.

2. Session Keys:
   The cryptographic primitives are regenerated on a relatively frequent basis. The key sequence on the inversely read list is unverifiable to an intruder not in possession of

the 'Autokey-value' message. Unnecessary 'bogus signature' checks are prevented by timestamp checks.

3. Autokey protocols:
   The nature of the protocols ties each message to its immediate former message and ultimately to the cryptographic primitives. Recovery from lost messages is enabled by possession of the values sent in the 'Autokey-value' message at key list generation time.

4. NTP Timekeeping Algorithm:
   Additional security is obtained by the ten sanity** checks performed by the NTP timestamp evaluation algorithms in the determination of a suitable source for synchronization.

3. NTP Modes:

3.1 Client/Server Mode:

3.1.1 Overview:

The dialogue begins with a Cookie request sent from the Client to the Server. The Server responds with an RSA signed message containing the Cookie. Note that the Server does not keep the Cookie, as the Server can regenerate the Cookie as required. The Client stores the Cookie and builds a Session Key list. From this point onward, NTP protocol messages between the Client and Server are moderated by Key-ID, (K_{id}), values, (which change with every message), and the Cookie. A Client poll request uses a session key in calculating the Auth. MAC. The Server recalculates the Auth.MAC to validate the Client and the Client maintains message request/response synchronization by comparing the K_{id} in the response to the K_{id} in the request.

3.1.2 Dialogue Details:

The Client builds a cookie request message using the standard request template, and sends it to the Server.

The Server rolls a 32 bit random number, (which it saves for use with all Clients), and creates the Cookie as the first 4-octets of the hash, (hash=MD5 ($|ip_{client}|ip_{server}|k_{id}=0|N_{random}|$)). The Cookie is sent to the Client in a cookie response message.

The Client verifies the signature using the Server public key, which was obtained either outside NTP or from the Server via a 'hostname/public key' request message. The Client checks the Server timestamp for a valid non-zero value, (a value of zero is a flag that the putative Server is not synchronized and should not be used until it is synchronized to its own sources).

The Client saves the Cookie for use in poll requests and validations.

The Client generates a key list using the cookie, (note that the key list is only generated on the Client).

The Client enters the main timekeeping dialogue loop:

1. START: The Client builds a poll request NTP message to be sent to the Server with no extension fields, calculates the MAC=MD5(|$K_{session}$|NTP header fields) using a session key from its key list and builds the Auth fields; Auth.K_{id}= $K_{id\ and}$ Auth.MAC=MAC.

2. The Server retrieves its random number and re-calculates the cookie.
   The Server verifies the MAC by reconstructing the key using the K_{id} from Auth.K_{id}, the re-generated cookie, and the Client and Server IP addresses:
   K_{test} = MD5(|S_{ip}|D_{ip}|K_{id}|cookie|). It then uses the Test-Key to hash the received NTP header fields: MD5(K_{test}|NTP header fields|) and compares the result with Auth.MAC.

3. Assuming the result of the hash comparison is valid, the Server builds an NTP timestamp response message for the Client using an NTP timestamp and an Authenticator, Auth.K_{id}=K_{id} using the K_{id} sent by the Client, and an Auth.MAC built from the Client's K_{id}, Cookie, and the ip addresses of the Client and Server as follows: $K_{session}$= MD5(|S_{ip}|D_{ip}|K_{id}|cookie). Auth.MAC = (MD5(|$K_{session}$|NTP-response header-fields|). Note that in the Client/Server mode the Server does not build a key list. However, the session key, $K_{session,}$ is different in both directions as the Client and Server Ip addresses are permuted in the $K_{session}$ hashing process.

4. The Client extracts the K_{id} from Auth.K_{id} and verifies that it matches the K_{id} of the request. If it does match, the Client calculates $K_{session}$: MD5(|S_{ip}|D_{ip}|K_{id}|cookie|) and uses $K_{session}$ to validate Auth.MAC: MAC = MD5(|$K_{session}$|NTP response-header-fields|). Assuming everything is correct the Client saves the timestamp. The timestamp information is used in the main NTP timekeeping algorithm to determine the dynamic suitability of the Server based on dispersion** and diffusion** calculations after compensation for network delay.

5. At the next Client determined poll interval, the Client loops to the start of the poll dialogue to obtain another timestamp from the Server.

6. The Client-Server dialogue continues in this loop until the Client key list is exhausted, at which point another key list is generated and the loop is reentered with the new key list.

When the Server regenerates the random number the cookie is derived from, the Server will fail to authenticate subsequent Client requests and will return a 'NAK' response, (i.e. the K_{id} will be equal to zero). The Client ignores messages with the K_{id}='NAK' and eventually the Association will encounter a watchdog timeout and the protocol will be restarted.

3.2 Symmetric (Peer) Modes:

3.2.1 Overview:

Either Peer may take the part of the Server or the Client. However, this relationship can change if the root synchronization distance** of the respective Peers' changes.

The Symmetric Mode has two sub-modes, Active/Active and Active/Passive.

Active/Active Sub Mode:

This mode's dialogue is brief, as the peer relationship has been pre-configured on both Peers and each Peer will have the opposite Peer's hostname, RSA public keys and Diffie-Hellman parameters available as files. All that is required by the dialogue is to exchange DH public-keys, generate the shared secret, build individual key lists, verify timestamps, calculate network delays, and enter the poll loop.

Active/Passive Sub Mode:

The dialogue is started by the Active Peer, which must exchange Canonical Hostname, RSA Public Keys, DH Parameters, and DH Public Keys in a preliminary dialogue before engaging in a dialogue essentially the same as that of the Active/Active mode dialogue.

3.2.2 Dialogue Details:

Preliminary Dialogue:

The Active/Passive Mode preliminary dialogue is started by the Active Peer:

1. Active Peer: Obtain the Passive-Peer's canonical hostname and RSA public key, (if not already obtained from other sources), and negotiate a common set of DH parameters with the Passive Peer. Build a key list using the default key and cookie, of zero, and send its DH public key in an RSA signed extension field message to the Passive Peer.
2. Passive Peer: Obtain the hostname and RSA public key for the Active Peer, either via a request message to the Active Peer or from a trusted source such as a Certificate Server or Secure DNS. Use the RSA public key to verify the signature in the received packet's extension field, and then use the common DH parameters to construct a set of DH keys, calculate the shared secret, and build the common cookie from the first 4-octets of the shared secret. The response packet sent to the Active Peer contains the Passive-Peer's DH public value, timestamp, and a request for the Active Peer's 'Autokey-values' in signed extension fields.

Common Pre-Poll Dialogue:

Both sub modes are essentially the same. However, it in the interests of continuity, the description will continue to use the Active/Passive labels:

3. Active Peer: Calculate the common cookie using the Passive's DH public key, and build a new key list using the cookie which is the first 4-octets of the shared secret. It then constructs a timestamped and signed 'Autokey-value' message to send to the Passive Peer, along with a request for the 'Autokey-values' of the Passive Peer.
4. Passive Peer: Build a key list using the common cookie and send a signed, timestamped 'Autokey-value' message to the Active Peer and set the authentic bit in the Passive Peer's status word.
5. Active Peer: Check the received signatures and timestamps store the Passive Peer's 'Autokey-value' and set the authentic bit in the Active Peer's status word.
6. The two Peers now dispense with extension fields and exchange timestamps to calibrate the delays. The Peer with the shortest root synchronization distance[**], (lowest stratum number), becomes the Server and the other becomes the Client. The Client then sets its clock and the normal polling sequence begins.

The peers now enter the main timekeeping poll loop. Since which peer will act as Client, and which will act as Server is dynamic dependent upon current stratum, this description will assume that the Active Peer is the Client and the Passive Peer is the Server.

Poll Loop:

8. START: The Active Peer builds a poll request message to be sent to the Passive Peer using a session key from its key list for the Auth fields; Auth.K_{id}= K_{id} and Auth.MAC=MD5(|$K_{session}$|NTP-header-fields|).

9. The Passive Peer verifies the MAC by first extracting K_{id} from Auth.K_{id}, and re-creating the Active peer's session key, $K_{session}$ = MD5(|S_{ip}|D_{ip}|K_{id}|Cookie|). It confirms that the received K_{id} is the next expected K_{id} from the Active Peer, and then uses the $K_{session}$ key to hash the received NTP header fields: MD5($K_{session}$|NTP-header-fields|) and compares the result with Auth.MAC.

10. Assuming the result of the hash comparison is valid, the Passive Peer builds a poll response message for the Active Peer including an NTP timestamp and a MAC built from the next session key from its key list; Auth.K_{id}=K_{id} and Auth.MAC = MD5(|$K_{session}$|NTP-header-fields|).

11. The Active Peer verifies the received message in the same manner as that described for the Passive Peer in step 9 above. Assuming everything is correct

the Active Peer saves the timestamp.

The timestamp information is used in the main NTP timekeeping algorithm to determine the dynamic suitability of the Passive Peer as a synchronization source based on dispersion** and diffusion** calculations after compensation for network delay.

12. At the next Active Peer determined poll interval, the Active Peer loops to the start of the poll loop to obtain another NTP timestamp from the Passive Peer.

13. The 'Active Peer-Passive Peer' dialogue continues in this loop until one of the peers' key list is exhausted, at which point that peer generates and distributes a new key list and the poll loop is reentered.

In the event of a lost 'Autokey-value' message, or a refresh of the DH public key by either peer, the recipient peer will fail to authenticate subsequent partner peer messages and will return a 'NAK' response, (i.e. the K_{id} will be equal to zero). Authentication will fail and the watchdog will timeout.

The protocol requires that when a Peer receives a DH public key value resulting in a different cookie it must respond with its own DH public key value. Both peers can now re-generate their key lists based on the new common Cookie developed from the new shared secret.

3.3 Multicast Mode:

Overview:

There are three modes grouped under Multicast; Broadcast, Multicast, & Manycast.

The essential differences between Multicast and Broadcast modes is in the IP address space used and the range of the advertisements.

Manycast operates in the Multicast IP range but the client actively 'trolls' for suitable servers rather than listening passively for advertisements.

The remainder of this discussion will focus on Multicast.

The Server regularly sends unsolicited advertisement messages at a rate of about one per minute. These messages always contain an extension field, either the 'Association-Id' extension field or, if the Server has just regenerated its key list, the 'Autokey-value' extension field.

Note that in a departure from the norm, the multicast 'Association-Id' extension message is transmitted unsigned in the advertisement messages. However, the 'Autokey-value' message is signed in the normal manner.

Since the Multicast advertisements always contain an extension field, the cookie is always zero and the session key, $K_{session}$ is always based on public values. The Association-Id identifies the Server to all Clients.

The Client, on receipt of the Server advertisement, changes to Client-Server Mode to authenticate the Server, obtain a valid timestamp and set the authentic bit in the Client's status word. After authentication, the Client will remain in Client-Server Mode long enough to gather enough timestamps to calibrate the Server delays

The Client then switches to Multicast-Client Mode and does not send again unless the Server happens to regenerate the key list and the Client misses the 'Autokey-value' message. If this happens, the Client fails to authenticate the Server's advertisements and eventually undergoes a watchdog reset, which starts the protocol over again.

Dialogue Details:

Since the Server identity is unknown to the Client until the Client receives an advertisement, the Client must first obtain the Server's canonical hostname and RSA Public key from the Server using a 'hostname-public key' request message.

After validation, the Client must obtain the Server's key list, if it had not already been obtained from a Server advertisement, by building an 'Autokey-value' request using the Server's Association-id advertisement message as a request handle. After this dialogue is complete, the Client goes into Multicast-Client Mode and just listens to the Server's Multicast advertisements to gather timestamps and refreshed 'Autokey-values'.

4 Installation & Configuration:

This focus of this discussion is on installation and configuration for the Solaris 2.6 Ultra/Sparc platforms and the Linux x86 platform running Red Hat 7.1 with the 2.4.2-2 kernel using the GCC compiler.

4.1 Installation:

In general terms, compiling and installation of NTP follows the well known './configure -> make -> make check -> make install' paradigm. However, to use the Autokey features of the program it is necessary to obtain a copy of either the rsaref20 library or the rsaeuro library and place it in the ntp directory tree. This is not as straightforward as it could be, for RSA no longer makes rsaref20 available at their web site and other locations on the Internet have several versions of both packages available.

After obtaining a copy of the library, 'untar' it into either an 'rsaref2' or 'rsaeuro1' directory in the NTP directory tree root, (see the README.rsa file for details). Then patch the source/rsa.c file in the library bundle using the patch found in the README.rsa file, which came

with the NTP distribution. Unfortunately, not all versions of either library are patchable, so it may be necessary to try several versions to locate one that is patchable. This is an important point, for if a version of the library is used which has to be patched by hand, there is no assurance that other modifications have not been made which will render the library incompatible with the NTP source code in possibly subtle ways.

'./configure' will find the library and build the Makefile with no problems. However, 'make' may exit with an error. If this happens there is an incompatibility in the rsa library and a different library will have to be located. Eventually, a library will be found that can be both patched and compiled.

After 'make' has run successfully, run 'make check' and 'make install'.

4.2 Configuration:

Configuration consists of two steps, creation of the ntp.conf file and generation of the required keys. The first step is to create the ntp.conf file. This should be done first as the key generation utility, ntp-genkeys, reads the configuration file when it runs.

The writer has built three configurations for the client-server, active-peer, and passive-peer modes of operation respectively. The only option added was logging. The configuration files used are in Addendum 9: Configuration Files.

In general, the configurations are straightforward but complex in the number of available options and require a **close reading** of the configuration and ntpd html pages of the distribution. The daemon will ignore and log any misunderstood configuration commands.

The final step is to generate the keys. There is a bug in the ntp-genkeys utility if it is run with no qualifiers, which will cause it to place all the keys in the root directory with broken links in the /etc/ntp and /usr/local/etc directories.

The writer has found the best way to generate keys is to cd to the /usr/local/etc directory and run 'ntp-genkeys –h' from there. This will create all of the keys in the /usr/local/etc directory and will not create any links. The keys which are generated are ntp.keys.<fs>, ntpkey.<fs>, ntp_dh.<fs>, and ntpkey_hostname.<fs> where <fs> is the common ntp timestamp used as a file stamp to identify the key series. The keys are the md5 keys, (NTP-V3 symmetric keys), RSA private key, Diffie-Hellman agreement parameters, and the RSA public key respectively. Move ntp.keys.<fs> and ntpkey.<fs> to the /etc/ntp directory.

Create links as follows:
In /usr/local/etc: ntp_dh -> ntp_dh.<fs> and ntpkey_hostname -> ntpkey_hostname.<fs>.
In /etc/ntp: ntp.keys -> ntp.keys.<fs> and ntpkey -> ntpkey.<fs>

To run in peer mode between active peers it is essential that both peers share a common

ntp_dh.<fs> file. It does not make any difference which peer's Diffie-Hellman parameter file is used, but the peers must share a common file. If both peers have a different file, the file with the newest file stamp will be used. If one peer does not have a parameter file, it can obtain it from the other with an Autokey message exchange. Alternatively, a copy of the file can be copied from one peer to the other and placed in /usr/local/etc linked to ntp_dh

One additional file needed is the leap-seconds.<fs> file available by ftp from the NIST timeservers, (e.g. ftp://time-b.nist.gov/pub/leap-seconds.list), and place it in /usr/local/etc with a link ntp_leap -> leap-seconds.<fs>. Note that the entry 'leap-seconds.list' is a link to the current version of the leap-seconds.<fs> file.

Some last points:

Do not create an empty 'drift' file in /etc/ntp, the daemon will create the file itself when it has garnered enough information to calculate drift.

Do create the logging directory and file, (e.g. /var/log/ntp/ntp-log).

Start the daemons; 'sudo ntpd' will work provided the configuration file ntp.conf is in the default '/etc' directory. Daemon operation may be monitored by observing the ntp-log file, or by using either of the two utilities, ntpq or ntpdc.

4.3 Observations and Comments:

Solaris:

NTP 4 performs as expected in the modes which were tried, (Client-Server, & Peer-Peer), when running on Solaris 2.6 machines, (note that it was **not** tried on x86 platforms, only on sparc and ultra platforms). One instance of the daemon runs which forms three sockets; *.123, 127.0.0.1.123, and <host-address>.123.

Linux (R.H. 7.x):

2.2.16 Kernel: NTP compiles with no fatal errors but cannot run successfully. It appears that the child process, which should hook the required UDP-123 port, dies when it is forked.

2.2.18 Kernel: NTP compiles and the daemon starts. However, the three children it spawns cannot hook the required ports unless the daemon is started in debug mode. When running in debug mode, the three children form three sockets, 127.0.0.0:123, 0.0.0.0:123, and <host-address>:123. Client-Server mode operates successfully, but Peer modes have problems maintaining Autokey authentication.

2.4.2 Kernel: NTP compiles and the daemon starts normally. It spawns three children, which form the same three sockets as with the 2.2.18 Kernel. Client-Server mode operates successfully, but the Peer modes have the same problems as the 2.2.18 Kernel.

Hopefully, the Linux issue will be addressed in future releases.

5. Addenda:

5.1 Addendum 1 – Overview of NTP

NTP is used to contact a group of, (typically three), timeservers on the Internet via the IANA assigned port UDP 123 and, by means of the NTP timekeeping algorithm, dynamically select a timeserver from the group to use as a reference for the synchronization of the host system clock.

NTP synchronizes clocks on the Internet in a hierarchical arrangement, (termed Strata), where Stratum-0 is a time standard, Stratum-1 is a host machine directly connected to a time standard, Stratum-2 is a host obtaining its time via NTP from a Stratum-1 machine, etc.

NTP is based on the receipt of timestamps and the calibration of network delays derived from these timestamps. Timestamps are obtained via polls using a 'request/response' message exchange with a Server in all modes other than Multicast, where timestamp messages are advertised at regular intervals by the Server and the Clients just passively 'listen'.

In NTP-V4 message fields are used to pass association-id and cryptographically related information between Servers and Clients.

NTP-V4 Timestamp flow can be shown as follows: server time -> delay for cryptographic operations -> delay of delivery to the network -> network delay -> delivery delay to the NTP application -> delay for cryptographic validation -> recipient host NTP timekeeping process. The largest processing time uncertainty is in the cryptographic processes, which can range from 1 – 10 μ sec for MD5 to hundreds of milliseconds for RSA dependent on the host architectures.

On the Client, the NTP clock filter selects the best candidates from a cluster of potential Servers via a software frequency/phase-lock loop based on the NTP timekeeping algorithm,

which minimizes error** and drift**.

The overall NTP filter algorithm errors range from < 1ms on LANS to ~10ms on WANS

The TAI leap second table, which is used to correct UTC to Atomic clock time, can either be obtained outside the NTP protocols or via an Autokey protocol request.

For additional NTP timekeeping details see the reference section of this paper.

5.2 Addendum 2 - Session Key Generation:

The key list is an array of keys used in the NTP authentication protocols. The keys are identified by a K_{id} number. K_{id} numbers have a total range of $1 – 2^{32}$, with NTP-V3 symmetric keys occupying the range of $1 \rightarrow (2^{16} – 1)$ and NTP-V4 session keys occupying the range of $2^{16} \rightarrow (2^{32}-1)$.

The remainder of the discussion focuses on the Autokey portion of the list.

Session keys are generated and placed into a key list. There is one key list for each Association maintained by the host.

The key list entry is a session key consisting of the Key-ID and the 128-bit hash (MD5 ($|S_{ip}|D_{ip}|K_{id}|Cookie|$)) Where S_{ip}=Source IP, D_{ip}=Destination IP, K_{id} = Key-ID.

The Key-ID can have three values:
$K_{id}(1^{st})$ = a Random number, N_{random}, where $2^{16} \leq N_{random} \leq (2^{32}-1)$.
$K_{id}(n)$ = a 4-octet pseudo-random number equal to the first 4-octets of the previous key hash that was stored in the list.
$K_{id}(NAK) = 0$

While the key list is being generated, if there is a collision between a K_{id} value already used on the list, or the K_{id} value is outside of the allowable range, the list will be deleted and re-generated from the beginning.

The size of the key list is a function of the poll interval, with approximately one hour's worth of keys being generated. The default minimum poll interval is 64 seconds and the default maximum poll interval is 1024 seconds. However, the absolute maximum number of session keys is determined by the value of NTP_MAXSESSION at compile time. In the reference distribution this value is set to 100.

NTP-V3 keys have a permanent lifetime, while an Autokey session key lifetime is one poll interval after it is scheduled to be used.

The key list is used in reverse order with the last key generated being the first key used.

Therefore, the recipient can validate the K_{id} of the current session key by testing that the first 4-octets of the current $K_{session}$ are equal to the K_{id} of the previous session key.

An example list:

$I_{dx}0$: $K_{id}(1^{st})|MD5(|S_{ip}|D_{ip}|K_{id}(1^{st})|cookie|)$
$I_{dx}1$: $K_{id}(1)|MD5(|S_{ip}|D_{ip}|K_{id}(1)|cookie|)$
$I_{dx}2$: $K_{id}(2)|MD5(|S_{ip}|D_{ip}|K_{id}(2)|cookie|)$
$I_{dx}3$: $K_{id}(3)|MD5(|S_{ip}|D_{ip}|K_{id}(3)|cookie|)$
$I_{dx}4$: $K_{id}(4)|MD5(|S_{ip}|D_{ip}|K_{id}(4)|cookie|)$ ← First key used.

'Final I_{dx}=4', 'Next K_{id} '= first 4 octets of session key located at $I_{dx}4$, (i.e. the value of $K_{id}(5)$ if such a key had been generated).

After the list is generated, the 'Autokey-value' message is built including the 'init-seq' field equal to the 'Final I_{dx} 'value from the example and the 'init-key-id' field equal to the 'Next K_{id}' value from the example.

The signed 'Autokey-value' message is sent unsolicited to the partner, or in the case of Multicast is sent in Multicast advertisement message. Since the 'Final I_{dx}' points to the first K_{id} ' used, and the current $K_{session}$ is linked to the previous K_{id}, all keys on the key list are traceable to the RSA Signature associated with the first key used from the list.

The recipient can now validate the first MAC generated by the Server using the key list as 'Next K_{id} '=first 4 octets of the first $K_{session}$ key that is used.

Key lists are purged under the following circumstances:

1. A switch from Client-Server to Multicast-Client Mode, (the key list is no longer needed)
2. A change in the poll interval, (the expiry time of the keys is no longer valid).
3. A general reset occurs.
4. There is an error in fetching a key from the list.
5. There is a refresh of DH private and public keys, (all Client association key lists are purged).
6. The Client is first synchronized, i.e. the clock is stepped**, (all Client association key lists are purged).

Key List Usage Examples:

Here are two examples of the use of the key list. The first example is the use of the list where all messages are received. The second example demonstrates list auto-recovery in the presence of a missed message.

Note: This example is taken from fragment of an ntpd debug dump. The addresses have been

sanitized.

Preliminary: Construct a key list: (this excerpt shows the construction of the last 10 keys generated)

Note: The 'session_key' value is not shown, as the debug code only shows the key-id values not the full md5 hash of 'K$_{session}$ = MD5(|S$_{ip}$|D$_{ip}$|K$_{id}$|Cookie|)'. See the session_key routine in ntp_crypto.c for full details.

start of edited dump extract
#fields: src-addr dest-addr key-id cookie nxt key-id sec
session_key: 192.168.10.5 > 192.168.1.1 9fdb7143 403d41d0 hash 6c3e34ae life 1408
session_key: 192.168.10.5 > 192.168.1.1 6c3e34ae 403d41d0 hash 60d87fce life 1280
session_key: 192.168.10.5 > 192.168.1.1 60d87fce 403d41d0 hash 9bffcf42 life 1152
session_key: 192.168.10.5 > 192.168.1.1 9bffcf42 403d41d0 hash 1672e1fe life 1024
session_key: 192.168.10.5 > 192.168.1.1 1672e1fe 403d41d0 hash 02f659d0 life 896
session_key: 192.168.10.5 > 192.168.1.1 02f659d0 403d41d0 hash cc5a347b life 768
session_key: 192.168.10.5 > 192.168.1.1 cc5a347b 403d41d0 hash 610494bc life 640
session_key: 192.168.10.5 > 192.168.1.1 610494bc 403d41d0 hash 7935a380 life 512
session_key: 192.168.10.5 > 192.168.1.1 7935a380 403d41d0 hash 5dbee5a5 life 384
session_key: 192.168.10.5 > 192.168.1.1 5dbee5a5 403d41d0 hash 5efa4629 life 256
#fields: seq key-id cookie timestamp exp
make_keys: 10 5efa4629 403d41d0 ts 3204571369 poll 7
end of edited dump extract

The 'seq' and 'key-id' will become the 'Init-Seq' and 'Init-KeyId' fields respectively of the 'Autokey value' message.

Construct the 'Autokey-value' message extension field:

|flags|code='autokey-value'|field-length|Assoc-Id|T.S.|Init-Seq=10|Init-KeyId=5efa4629|Sig-len|Sig|

The 'Autokey-values' are sent to the partner.

Notes:

In the following descriptions, the md5 hashes are **artificial** and are not what would be obtained by hashing the purported values for two reasons; the addresses have been sanitized and the data format passed to md5 is not the format displayed by the debug message. See ntp_crypto.c for details.

Additionally, the variables 'exp-hash-frag' and 'count-seq' are constructs of the example used to illustrate the intent of the code and are not to be taken as an explicit definition of how these functions are actually accomplished within the code.

Normal Messaging:

An example of a normal message transmission sequence as viewed by the Recipient:
Recall that the keys are used from the list in the inverse order of their generation.

Copy Init-Key-Id from the Autokey-Value message to a variable, say exp-hash-frag.
exp-hash-frag=Init-Key-Id=5efa4629.
Save Init-Seq as count-seq.
count-seq=Init-Seq=10

Message-1:
Authenticator Fields: |K_{id}=5dbee5a5|MD5-hash-of-NTP-header-fields|
Construct Session Key:
$K_{session}$=MD5(192.168.10.5192.168.1.15dbee5a5403d41d0)=
5efa4629ff5bca6493ea27feef1e31b3
Comparison of 'exp-hash-frag' and first 4 octets of $K_{session}$: 5efa4629 = 5efa4629
Save 'exp-hash-frag'=K_{id} =5dbee5a5
Decrement count-seq.

Message-2:
Authenticator Fields: |K_{id} =7935a380|MD5-hash-of-NTP-header-fields|
Construct Session Key:
$K_{session}$ =MD5(192.168.10.5192.168.1.17935a380403d41d0)=
5dbee5a52040bf60ba2ba66b897f94d0
Comparison of 'exp-hash-frag' and first 4 octets of $K_{session}$: 5dbee5a5=5dbee5a5
Save 'exp-hash-frag'=K_{id} =7935a380
Decrement count-seq.

Message 3:

Authenticator Fields: |K_{id} =610494bc|MD5-hash-of-NTP-header-fields|
Construct Session Key:
$K_{session}$ =MD5(192.168.10.5192.168.1.1610494bc403d41d0)=
7935a380e10e1f8be558df7cfad28fee
Comparison of 'exp-hash-frag' and first 4 octets of $K_{session}$: 7935a380=7935a380
Save 'exp-hash-frag'=K_{id} =610494bc
Decrement count-seq.

Message 4:

Authenticator Fields: |K_{id} =cc5a347b|MD5-hash-of-NTP-header-fields
Construct Session Key:
$K_{session}$ =MD5(192.168.10.5192.168.1.1cc5a347b403d41d0)=
610494bcc6401ad6901465b4fe235417
Comparison of 'exp-hash-frag' and first 4 octets of $K_{session}$: 610494bc=610494bc

Save ‘exp-hash-frag’=K_{id} = cc5a347b
Decrement count-seq.

Missed Message Auto-recovery:

An example of auto-recovery of the key list due to a missed message as viewed by the Recipient

Copy Init-Key-Id from the Autokey-Value message to a variable, say exp-hash-frag.
exp-hash-frag=Init-Key-Id=5efa4629.
Save Init-Seq as count-seq.
count-seq=Init-Seq=10

Message-1:
Authenticator Fields: |K_{id}=5dbee5a5|MD5-hash-of-NTP-header-fields|
Construct Session Key:
$K_{session}$=MD5(192.168.10.5192.168.1.15dbee5a5403d41d0)=
5efa4629ff5bca6493ea27feef1e31b3
Comparison of ‘exp-hash-frag’ and first 4 octets of $K_{session}$: 5efa4629 = 5efa4629
Save ‘exp-hash-frag’=K_{id} =5dbee5a5

Message-2:

Missed Message

Message 3:

Authenticator Fields: |K_{id} =610494bc|MD5-hash-of-NTP-header-fields|
Construct Session Key:
$K_{session}$ =MD5(192.168.10.5192.168.1.1610494bc403d41d0)=
7935a380e10e1f8be558df7cfad28fee
Comparison of ‘exp-hash-frag’ and first 4 octets of $K_{session}$: 5dbee5a5≠7935a380

Auto-recovery:

Compute a Test-Session key using the first 4 octets of the unexpected hash fragment as a session key, K_{id}
$K_{session}$test =MD5(192.168.10.5192.168.1.17935a380403d41d0)=
5dbee5a52040bf60ba2ba66b897f94d0
Decrement count-seq

Compare ‘exp-hash-frag’ with the first 4 octets of the test session key: 5dbee5a5=5dbee5a5.

Since they equate, The Session key with the first 4-octet sequence is in the key list and the ‘exp-

hash-frag' sequence was the key-id of a missed message.

Therefore, the Session Key with the K_{id} value of 610494bc is a valid key and may be used for checking the MAC. Additionally 'exp-hash-frag' can be reset to 610494bc, which will restore the list from the perspective of the Recipient.

Save 'exp-hash-frag'=K_{id} =610494bc
Decrement count-seq.

Post-recovery:

Message 4:

Authenticator Fields: |K_{id} =cc5a347b|MD5-hash-of-NTP-header-fields
Construct Session Key:
$K_{session}$ =MD5(192.168.10.5192.168.1.1cc5a347b403d41d0)=
610494bcc6401ad6901465b4fe235417
Comparison of 'exp-hash-frag' and first 4 octets of $K_{session}$: 610494bc=610494bc
Save 'exp-hash-frag'=K_{id} = cc5a347b
Decrement count-seq

Multiple missed message keys can be recovered in this manner by iteration of the tests back to the value obtained as the 'Init-Key-id' in the 'Autokey-value' message with an iteration limit of count-seq=0.

However, if the failure was due to a cookie or key list change which went undetected the list will not be able to be reconstructed barring a MD5 hash collision which can be disregarded due to the nature of the MD5 function.

5.3 Addendum 3 – Cookies:

The 4-octet cookie can have multiple values based on NTP mode and Message type:

Cookie values:

1. Multicast Mode and **any** packet with an Extension Field = Zero (0)

2. Client-Server Mode = first 4-octets of the hash=MD5(|S_{ip}|D_{ip}|0|N_{random}|), where N_{random} is a Server generated 4octet random number.

3. Symmetric (Peer) Modes: = first 4-octets of DH_{secret}, where DH_{secret} is negotiated between the Peers.

5.4 Addendum 4 - NTP Message Format:

The NTP message is composed of a header section, optional extension sections, and an authenticator section. While any message may contain extension fields, usually poll messages do not contain extension fields.

1. NTP Header Format:

   The basic header is 48 octets in size, composed of 32 bit words containing identifier, delay, dispersion** and timestamp data

2. Extension Field Format:

   Extension fields are used for the transmission of association-id and cryptographically related information between hosts in the form of Autokey request/response information such as cookies, RSA public keys, DH parameters and negotiations. Extension response fields are always signed with the special exception of the Multicast 'Association-Id' advertisement messages.

   The extension fields consist of flag, type, length, association, data & padding. Padding is to 32 bit for any field except the last, which is padded to 64 bit.

3. Authenticator Field Format:

   The authenticator consists of multiple 32-bit fields, the first of which contains either algorithm identity, (NTP-V3), or the Key-ID. The remaining fields contain either the symmetric encryption of the Basic header, (NTP-V3), or the Keyed MD5 hash of the NTP Header fields plus any Extension fields.

   For detailed field information for these formats see Addendum 8.

5.5 Addendum 5 – Key Files:

In configured associations, the Server or Peer public key can be obtained from a file on the Client, which was obtained outside of the protocol, or the Client can obtain the public key from the Server via the NTP protocol.

In Multicast or Symmetric Passive association, the Server public key must be obtained from the Server via the NTP preliminary protocol, or from a trusted 3rd party source such as a Certificate Server or Secure DNS.

The following key files will be found on all machines:
Note <fs> = A file timestamp in NTP timestamp format

1. ntp.keys.<fs> = NTP-V3 symmetric keys.
   ntp.keys are generated with the other keys but not used by Autokey.
2. ntp_dh.<fs> = Diffie-Hellman Parameters.
   ntp_dh must be common amongst the constellation of machines but can be distributed by insecure means.
3. ntpkey_<hostname>.<fs> = RSA Public Key.
   ntpkey_<hostname>.<fs> can be distributed by insecure means.
4. ntpkey.<fs> = RSA Private Key.
   ntpkey must be kept secret on each machine.
5. ntpkey_certif_<hostname>.<fs> PKI Certificate.
   Currently obtained if available but not used.

These files are commonly linked to generic names. For example: ntp.keys -> ntp.keys.<fs>. Key files are created with the utility ntp-genkeys as a 'key-generation' with a common <fs> value. Therefore, new keys can be put in service merely by changing the files pointed to by the generic names.

5.6 Addendum 6 – Hostname & RSA Public Key Acquisition:

A Server's or Peer's RSA Public Key may be obtained in several ways; by means of a file transfer in the case of a configured association, (e.g. Active/Active Symmetric), from a trusted 3rd party source such as a Certificate Server or Secure DNS. Alternatively, it may be obtained directly from the Server with a 'hostname and public key request message' using the standard request template.

The Server responds with its canonical hostname, (defined as the return string from the UNIX gethostname () function), RSA Public Key, Public Key Timestamp, message timestamped and RSA signature.

If the signature cannot be verified, or the public key file cannot be found or loaded, the request is repeated until it is successful or there is a watchdog timeout.

The response contains the public key in network byte order, (big-endian) with the modulus being the first 32bit word and in the format defined in the RSAREF20 documentation.

5.7 Addendum 7 – Diffie-Hellman Values:

There are two sets of DH values that must be obtained. The DH parameters, 'N' & 'G',(where 'N' is a large prime and 'G' is a generator of 'N'), and the DH Public Key for the partner calculated using the parameter values.

$(DH\text{-}Public_{partner} = G^{DH\text{-}Private\text{-}partner} \bmod N)$.

The DH public key used to compute the shared secret,

($DH_{secret} = DH\text{-}Public_{partner}^{DHprivate\text{-}local} \bmod N$).

The Parameter and Public Key values may be obtained, either by 'out of protocol' means from files in the case of a configured Active/Active association or, from the partner Peer by means of 'Autokey request-response' extension message pairs. The requests use the standard request template extension message with the appropriate code number while the timestamped, RSA signed responses are returned in the relevant response messages.

5.8 Addendum 8 – Message Format Details:

The formats detailed in this Addendum are the ones currently defined for the Reference implementation. However, since this is a 'work in progress' the format details of the various messages may change as the protocol evolves pending promulgation of the Standard.

For current format details please refer to the latest edition of the papers cited in the References Section.

Common Definitions and notations:

1. The field name subscript number is the field size in bits, with a subscript of 'x' being used where the field is of a variable length.
2. Fields are padded to their respective boundaries, with the last field in a message padded to its 64 bit boundary, to ensure the extension fields are multiples of the 32 bit standard size.
3. There are only 2 flag bits currently defined, Reply and Error.
4. The '|' character indicates field boundaries.

1. NTP Header:
   |li|vn|mode|stratum|poll|precision|root-delay$_{32}$|ref-ident$_{32}$|ref-timestamp$_{64}$|
   orig-timestamp$_{64}$|rx-timestamp$_{64}$|tx-timestamp$_{64}$|extension-1$_x$|extension-n$_x$|
   key/algorithm-ident$_{32}$|mac-digest$_x$|

   For the fields from 'li' through 'tx-timestamp' see NTP-V3, (rfc-1305), for definitions.
   The extension fields are optional message fields.
   The key/algorithm field is the K_{id} of the session key in NTP-V4.
   The mac-digest size is 128-bits in V4.
   The key/algorithm and mac-digest fields are collectively known as the 'Authenticator'.

1.1 NTP Timestamp format:
   |seconds$_{32}$|fraction$_{32}$|
   The total value is the elapsed seconds and fractions since 1-jan-1900.

2. Request message extension fields:

All extension response fields are signed and timestamped with the timestamp being the time of signing and included in the signed field range.

2.1 Extension field code numbers take the form of 0x10n and 0x810n where 'n' is the code number in request and response messages respectively.

Currently assigned code numbers:

1. Parameter Negotiation – reserved for future use.
2. Association ID.
3. Autokey Value.
4. Cookie.
5. Diffie-Hellman Parameters.
6. Diffie-Hellman Public Key.
7. Hostname & RSA Public Key.
8. Certificate.
9. TAI Leapsecond Table.

2.2 Standard Request Template:
|$flags_8$|$code_8$|$length_{16}$|$association\text{-}id_{32}$|
Used by all codes except where noted.

2.3 Code–1. Parameter Negotiation
|$code_{16}$|$length_{16}$|$association\text{-}id_{32}$|$timestamp_{32}$|$param\text{-}len_n$|$parameters_n$|
$RSA\text{-}sig\text{-}len_{64}$|$RSA\text{-}signature_{512}$|
This code is not in use and is reserved for future use. It is only included here for continuity.

2.4 Code-2. Association-Id Request & Response:
Request:|$flags_8$|$code_8$|$length_{16}$|$association\text{-}id_{32}$|$status\text{-}flags_{32}$|
Association-id = 0 on request and Server Assigned Number on Response.
Status-Flag values currently in use:
Bit 31=Autokey Enabled, bit 30=Public & Private Keys loaded,
Bit 29=DH Agreement parameters loaded, bit 28=Leap Second table loaded.

2.5 Code-3. Autokey Value Response:
|$flags_8$|$code_8$|$length_{16}$|$association\text{-}id_{32}$|$timestamp_{32}$|$init\text{-}seq_{32}$|$init\text{-}key\text{-}id_{32}$|
$RSA\text{-}sig\text{-}len_{64}$|$RSA\text{-}signature_{512}$|
init-seq is the index number of the last K_{id} added to the key list.
init-key-id is the first 4 octets of the last Session key added to the key list.
Signed fields are 'timestamp' through 'init-key-id'.

2.6 Code-4. Cookie
Request:
Standard Request Template, Association-Id=0.
Response:
|$flags_8$|$code_8$|$length_{16}$|$association\text{-}id_{32}$|$timestamp_{32}$|$cookie_{32}$|

RSA-signature-len$_{64}$|RSA-signature$_{512}$|.
Association-ID is a server assigned number.
'timestamp' is time when the Cookie was generated and signed.
Signed fields are 'timestamp' through 'cookie'.

2.7 Code-5. Diffie-Hellman (DH) Parameters:
|flags$_8$|code$_8$|length$_{16}$|association-id$_{32}$|timestamp$_{32}$|filestamp$_{32}$|
prime-len$_{64}$|prime$_{512}$|gen-len$_{64}$|generator$_{512}$|RSA-sig-len$_{64}$|RSA-signature$_{512}$|
filestamp is the timestamp when the parameters file was generated.
Signed fields are 'timestamp' through 'parameters'.

2.8 Code-6. Diffie-Hellman (DH) Public Key:
|flags$_8$|code$_8$|length$_{16}$|association-id$_{32}$|timestamp$_{32}$|filestamp$_{32}$|DH-Public-key-len$_{64}$
|DH-Public-key$_{512}$|RSA-sig-len$_{64}$|RSA-signature$_{512}$|
filestamp' is timestamp when the DH parameters file was generated.
Signed fields are 'timestamp' through 'DH-Public-key'

2.9 Code-7. Hostname & RSA Public key:
|flags$_8$|code$_8$|length$_{16}$|association-id$_{32}$|timestamp$_{32}$|filestamp$_{32}$|
RSA-Public-key-len$_{32}$|RSA-Public-key-modulus$_{32}$|RSA-Public-key$_{2048}$|
Hostname-len$_{32}$|Hostname$_n$|RSA-sig-len$_{64}$|RSA-signature$_{512}$|
Association-ID=0 as the hostname and RSA public key are properties of the Server and not related to a specific association.
Filestamp is the timestamp when the RSA public key was generated.
RSA-Public-key format follows that of RSAREF20.
Signed fields are 'timestamp' through 'hostname'

2.10 Code-8. Certificate:
|flags$_8$|code$_8$|length$_{16}$|association-id$_{32}$|timestamp$_{32}$|filestamp$_{32}$|
Certificate-len$_{32}$|RSA-sig-len$_{64}$|RSA-signature$_{512}$|
Association-ID=0 as the certificate is a property of the Server and not related to a specific association.
Signed fields are 'timestamp' through 'hostname'

2.11 Code-9 Leap Second Table:
|flags$_8$|code$_8$|length$_{16}$|association-id$_{32}$|timestamp$_{32}$|filestamp$_{32}$|table-len$_{32}$|
table$_n$|RSA-sig-len$_{64}$|RSA-signature$_{512}$|
Association-ID=0 as leap-second-table is not a property of any specific association.
Filestamp is the timestamp of the generation of the Leap second (TAI) file on the originating machine, usually an NIST machine.
Signed fields are 'timestamp' through 'leap-sec-table'.

5.9 Addendum 9-Configuration Files:

These configurations were used to test the Autokey authentication functions and are

not meant to be complete NTP configuration files. They were written to implement three modes of operation; Client-Server, Active-Active Peers, and Active-Passive Peers respectively. Several options have been omitted and reference should be made to the relevant HTML document pages before implementing NTP on production machines.

These files have been 'sanitized' using the reserved domain 'example.org' and the reserved address block 192.168.0.0.

9.1 Client -> Server Mode:

```
#ntp.conf
#06-jul-01
#test configuration for authentication tests.
#client name is client.example.org
#
#set up server arrangement with server-1.example.org and server-2.example.org
#IP addresses have been used here to avoid possible DNS cache poisoning pending the
#completion of secure DNS.
#
server 192.168.10.20 autokey
server 192.168.11.21 autokey
#
driftfile /etc/ntp/drift
#
keys /etc/ntp/ntp.keys
keysdir /usr/local/etc
#
# the crypto keyword is required, but its qualifiers are optional.
# the use of the 'dh' as the syntax for the qualifier for the Diffie-Hellman key is correct
# and was gleaned from the source code.
# the following two lines are actually one continuous line in the configuration file.
crypto privatekey /etc/ntp/ntpkey publickey /usr/local/etc/ntpkey_client.example.org dh
/usr/local/etc/ntpkey_dh  leap/usr/local/etc/ntp_leap
#
logfile /var/log/ntp/ntp-log
#
# note that the 'logconfig' syntax differs from the 'Miscellaneous Options' Html page in
# that the 'logconfig=' syntax generates an error message on startup
# but the 'logconfig ' syntax appears to work.
logconfig clockall +peerall +sysall +syncall
#
```

9.2 Active Peer <–> Active Peer Mode:

```
#ntp.conf
```

```
#02-jul-01
#test configuration for authentication tests.
# this machine's name is peer-1.example.org.
#
#set up peer arrangement between peer-1.example.org and peer-2.example.org.
# IP addresses have been used here to avoid possible DNS cache poisoning pending the
#completion of #secure DNS.
#
#Lower stratum servers are required for traceability to the stratum root as neither peer is a
#stratum 1 machine with a directly connected stratum 0 clock.
server 198.168.12.22
server 198.168.13.23
#
#set up peer arrangement with peer-2.example.org
peer 192.168.14.24 autokey
#
driftfile /etc/ntp/drift
#
keys /etc/ntp/ntp.keys
keysdir /usr/local/etc
#
# the crypto keyword is required, but the qualifiers are optional.
# the use of the 'dh' as the syntax for the qualifier for the Diffie-Hellman key is correct
# and was gleaned from the source code.
# the following two lines are actually one continuous line in the configuration file
crypto privatekey /etc/ntp/ntpkey publickey /usr/local/etc/ntpkey_peer-1.example.org dh
/usr/local/etc/ntpkey_dh leap /usr/local/etc/ntp_leap
#
logfile /var/log/ntp/ntp-log
#
# note that the 'logconfig' syntax differs from the 'Miscellaneous Options' Html page in
# that the 'logconfig=' syntax generates an error message on startup
# but the 'logconfig ' syntax appears to work.

logconfig clockall +peerall +sysall +syncall
#
```

9.3 Active<->Passive Peer Mode:

The configuration files are essentially the same as for active <-> active peer mode with the exception of the peer statement being only in the active peer's configuration.

The passive peer's RSA public key file and the Diffie-Hellman, (DH) parameter file can be placed in the active peer's /usr/local/etc with the appropriate links if they are available from other sources such as a public key server.

5. Notes and References:

5.1 Notes:

* The term Multicast is used herein to collectively refer to the three multiple access modes Multicast, Manycast,& Broadcast unless specified otherwise.

** For descriptions of these terms refer to RFC 1305.

Since this is a 'work in progress' the possibility exists of differences between what is found in the html pages, which come with the distribution, and similar information posted on the web site. Resolution of such conflicts can be obtained by reference to the CVS repository at http://maccarony.ntp.org/cgi-bin/cvsweb.cgi/, or in extreme cases, by reference to the distribution source code.

5.2 Published References:

1. Html pages in the reference distribution:
   ntp-4.0.99m-rc2/html/index.htm

2. Source code in the reference distribution:
   ntp-4.0.99m-rc2/ntpd

3. Authentication Options. David L.Mills mills@udel.edu
   http://www.eecis.udel.edu/~ntp/ntp_spool/html/authopt.htm

4. Autonomous Authentication. David L. Mills
   http://www.eecis.udel.edu/~mills/autokey.htm

5. Network Time Protocol, Security Model & Authentication Scheme. David L. Mills
   http://www.eecis.udel.edu/~mills/database/brief/autokey/autokey.pdf

6. Public Key Cryptography for the Network Time Protocol. David L. Mills
   Electrical Engineering Dept, University of Delaware, May 2000
   http://www.eecis.udel.edu/~mills/database/reports/pkey/pkeya.pdf
   http://www.eecis.udel.edu/~mills/database/reports/pkey/pkeyb.pdf

7. Public-Key Cryptography for the Network Time Protocol, (version 1). David L. Mills
   <draft-ietf-stime-ntpauth-01.txt> April 2001
   This reference is an ietf internet draft and is a 'work in progress'. This specific draft has an expiry date of October 2001.
   Required inclusion of the <draft-ietf-stime-ntpauth-01.txt> copyright statement:
   "Full Copyright Statement"
   "Copyright © The Internet Society (date). All Rights Reserved. This document and translations of it may be copied and furnished to others, and derivative works that comment

on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purposes of deploying Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into.

http://www.ietf.org./internet-drafts/draft-ietf-stime-ntpauth-01.txt

8. Cryptographic Authentication for Real-Time Network Protocols. David L. Mills
In: AMS DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 45 (1999), 135-144.
http://www.eecis.udel.edu/~mills/papers.htm

9. Network Time Protocol, Distribution Notes. David L. Mills
http://www.eecis.udel.edu/~ntp/ntp_spool/html/index.htm
Distribution Copyright Statement:
****
*
*
*
* Copyright (c) David L. Mills 1992-2001 *
*
*
* Permission to use, copy, modify, and distribute this software and *
* its documentation for any purpose and without fee is hereby *
* granted, provided that the above copyright notice appears in all *
* copies and that both the copyright notice and this permission *
* notice appear in supporting documentation, and that the name *
* University of Delaware not be used in advertising or publicity *
* pertaining to distribution of the software without specific, *
* written prior permission. The University of Delaware makes no *
* representations about the suitability this software for any *
* purpose. It is provided "as is" without express or implied *
* warranty. *
*
*
****
*

10. RFC 1305Network Time Protocol (Version 3) Specification, Implementation and Analysis
David L. Mills, University of Delaware, Mar 1992
http://rfc.net/rfc1305.html

12. Source of leap-seconds.<fs> file:
ftp://time-b.nist.gov/pub/leap-seconds.list
Note that the entry ‘leap-seconds.list’ is a link to the current version of the

leap-seconds.<fs> file.

Last Updated: July 27th, 2017

Upcoming Training

SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
Community SANS Omaha SEC401*	Omaha, NE	Aug 14, 2017 - Aug 19, 2017	Community SANS
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Salt Lake City 2017	Salt Lake City, UT	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
Virginia Beach 2017 - SEC401: Security Essentials Bootcamp Style	Virginia Beach, VA	Aug 21, 2017 - Aug 26, 2017	vLive
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
Community SANS Pasadena SEC401 @ NASA	Pasadena, CA	Aug 23, 2017 - Aug 30, 2017	Community SANS
Mentor Session - SEC401	Minneapolis, MN	Aug 29, 2017 - Oct 10, 2017	Mentor
SANS San Francisco Fall 2017	San Francisco, CA	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Tampa - Clearwater 2017	Clearwater, FL	Sep 05, 2017 - Sep 10, 2017	Live Event
Mentor Session - SEC401	Edmonton, AB	Sep 06, 2017 - Oct 18, 2017	Mentor
SANS Network Security 2017	Las Vegas, NV	Sep 10, 2017 - Sep 17, 2017	Live Event
Community SANS Albany SEC401	Albany, NY	Sep 11, 2017 - Sep 16, 2017	Community SANS
Mentor Session - SEC401	Ventura, CA	Sep 11, 2017 - Oct 12, 2017	Mentor
Community SANS Columbia SEC401	Columbia, MD	Sep 18, 2017 - Sep 23, 2017	Community SANS
Community SANS Dallas SEC401	Dallas, TX	Sep 18, 2017 - Sep 23, 2017	Community SANS
Community SANS New York SEC401	New York, NY	Sep 25, 2017 - Sep 30, 2017	Community SANS
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS London September 2017	London, United Kingdom	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Copenhagen 2017	Copenhagen, Denmark	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS Boise SEC401	Boise, ID	Sep 25, 2017 - Sep 30, 2017	Community SANS
Baltimore Fall 2017 - SEC401: Security Essentials Bootcamp Style	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	vLive
Community SANS Sacramento SEC401	Sacramento, CA	Oct 02, 2017 - Oct 07, 2017	Community SANS
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Community SANS Charleston SEC401	Charleston, SC	Oct 02, 2017 - Oct 07, 2017	Community SANS
Mentor Session - SEC401	Arlington, VA	Oct 04, 2017 - Nov 15, 2017	Mentor
Community SANS Indianapolis SEC401	Indianapolis, IN	Oct 09, 2017 - Oct 14, 2017	Community SANS