1. Introduction

A distributed network service requires reliable, ubiquitous and survivable provisions to prevent accidental or malicious attacks on the servers and clients in the network or the values they exchange. Reliability requires that clients can determine that received packets are authentic; that is, were actually sent by the intended server and not manufactured or modified by an intruder. Ubiquity requires that any client can verify the authenticity of any server using only public information. Survivability requires protection from faulty implementations, improper operation and possibly malicious clogging and replay attacks with or without data modification. These requirements are especially stringent with widely distributed network services, since damage due to failures can propagate quickly throughout the network, devastating archives, routing databases and monitoring systems and even bring down major portions of the network.

The Network Time Protocol (NTP) contains provisions to cryptographically authenticate individual servers as described in the most recent protocol NTP Version 3 (NTPv3) specification [11]; however, that specification does not provide a scheme for the distribution of cryptographic keys, nor does it provide for the retrieval of cryptographic media that reliably bind the server identification credentials with the associated private keys and related public values. However, conventional key agreement and digital signatures with large client populations can cause significant performance degradations, especially in time critical applications such as NTP. In addition, there are problems unique to NTP in the interaction between the authentication and synchronization functions, since each requires the other.

This document describes a cryptographically sound and efficient methodology for use in NTP and similar distributed protocols. As demonstrated in the reports and briefings cited in the references at the end of this document, there is a place for PKI and related schemes, but none of these schemes alone satisfies the requirements of the NTP security model. The various key agreement schemes [8], [13], [5] proposed by the IETF require per-association state variables, which contradicts the principles of the remote procedure call (RPC) paradigm in which servers keep no state for a possibly large client population. An evaluation of the PKI model and algorithms as implemented in the OpenSSL library leads to the conclusion that any scheme requiring every NTP packet to carry a PKI digital signature would result in unacceptably poor timekeeping performance.

A revised security model and authentication scheme called Autokey was proposed in earlier reports [10], [9]. It is based on a combination of PKI and a pseudo-random sequence generated by repeated hashes of a cryptographic value involving both public and private components. This scheme has been tested and evaluated in a local environment and in the CAIRN experiment network funded by DARPA. A detailed description of the security model, design principles and implementation experience is presented in this document.

Additional information about NTP, including executive summaries, briefings and bibliography can be found on the NTP project page linked from www.ntp.org. The NTPv4 reference implementation for Unix and Windows, including sources and documentation in HTML, is available from the NTP repository at the same site. All of the features described in this document, including support for both IPv4 and IPv6 address families, are included in the current development version at that repository. The reference implementation is not intended to become part of any standard that may be evolved from this document, but to serve as an example of how the procedures described in this document can be implemented in a practical way.

2. NTP Security Model

NTP security requirements are even more stringent than most other distributed services. First, the operation of the authentication mechanism and the time synchronization mechanism are inextricably intertwined. Reliable time synchronization requires cryptographic keys which are valid only over designated time intervals; but, time intervals can be enforced only when participating servers and clients are reliably synchronized to UTC. Second, the NTP subnet is hierarchical by nature, so time and trust flow from the primary servers at the root through secondary servers to the clients at the leaves.

A client can claim authentic to dependent applications only if all servers on the path to the primary servers are bone-fide authentic. In order to emphasize this requirement, in this document the notion of "authentic" is replaced by "proventic", a noun new to English and derived from provenance, as in the provenance of a painting. Having abused the language this far, the suffixes fixable to the various noun and verb derivatives of authentic will be adopted for proventic as well. In NTP each server authenticates the next lower stratum servers and proventicates (authenticates by induction) the lowest stratum (primary) servers. Serious computer linguists would correctly interpret the proventic relation as the transitive closure of the authentic relation.

t is important to note that the notion of proventic does not necessarily imply the time is correct. A NTP client mobilizes a number of concurrent associations with different servers and uses a crafted agreement algorithm to pluck truechimers from the population possibly including falsetickers. A particular association is proventic if the server certificate and identity have been verified by the means described in this document. However, the statement "the client is synchronized to proventic sources" means that the system clock has been set using the time values of one or more proventic client associations and according to the NTP mitigation algorithms. While a certificate authority (CA) must satisfy this requirement when signing a certificate request, the certificate itself can be stored in public directories and retrieved over unsecured network paths.

Over the last several years the IETF has defined and evolved the IPSEC infrastructure for privacy protection and source authentication in the Internet, The infrastructure includes the Encapsulating Security Payload (ESP) [7] and Authentication Header (AH) [6] for IPv4 and IPv6. Cryptographic algorithms that use these headers for various purposes include those developed for the PKI, including MD5 message digests, RSA digital signatures and several variations of Diffie-Hellman key agreements. The fundamental assumption in the security model is that packets transmitted over the Internet can be intercepted by other than the intended receiver, remanufactured in various ways and replayed in whole or part. These packets can cause the client to believe or produce incorrect information, cause protocol operations to fail, interrupt network service or consume precious network and processor resources.

In the case of NTP, the assumed goal of the intruder is to inject false time values, disrupt the protocol or clog the network, servers or clients with spurious packets that exhaust resources and deny service to legitimate applications. The mission of the algorithms and protocols described in this document is to detect and discard spurious packets sent by other than the intended sender or sent by the intended sender, but modified or replayed by an intruder. The cryptographic means of the reference implementation are based on the OpenSSL cryptographic software library available at www.openssl.org, but other libraries with equivalent functionality could be used as well. It is important for distribution and export purposes that the way in which these algorithms are used precludes encryption of any data other than incidental to the construction of digital signatures.

There are a number of defense mechanisms already built in the NTP architecture, protocol and algorithms. The fundamental timestamp exchange scheme is inherently resistant to spoof and replay attacks. The engineered clock filter, selection and clustering algorithms are designed to defend against evil cliques of Byzantine traitors. While not necessarily designed to defeat determined intruders, these algorithms and accompanying sanity checks have functioned well over the years to deflect improperly operating but presumably friendly scenarios. However, these mechanisms do not securely identify and authenticate servers to clients. Without specific further protection, an intruder can inject any or all of the following mischiefs.

The NTP security model assumes the following possible threats. Further discussion is in [9] and in the briefings at the NTP project page, but beyond the scope of this document.

1. An intruder can intercept and archive packets forever, as well as all the public values ever generated and transmitted over the net.

2. An intruder can generate packets faster than the server, network or client can process them, especially if they require expensive cryptographic computations.

3. In a wiretap attack the intruder can intercept, modify and replay a packet. However, it cannot permanently prevent onward transmission of the original packet; that is, it cannot break the wire, only tell lies and congest it. Except in unlikely cases considered in Appendix D, the modified packet cannot arrive at the victim before the original packet.

4. In a middleman or masquerade attack the intruder is positioned between the server and client, so it can intercept, modify and replay a packet and prevent onward transmission of the original packet. Except in unlikely cases considered in Appendix D, the middleman does not have the server private keys or identity parameters.

The NTP security model assumes the following possible limitations. Further discussion is in [9] and in the briefings at the NTP project page, but beyond the scope of this document.

1. The running times for public key algorithms are relatively long and highly variable. In general, the performance of the time synchronization function is badly degraded if these algorithms must be used for every NTP packet.

2. In some modes of operation it is not feasible for a server to retain state variables for every client. It is however feasible to regenerated them for a client upon arrival of a packet from that client.

3. The lifetime of cryptographic values must be enforced, which requires a reliable system clock. However, the sources that synchronize the system clock must be cryptographically proventicated. This circular interdependence of the timekeeping and proventication functions requires special handling.

4. All proventication functions must involve only public values transmitted over the net with the single exception of encrypted signatures and cookies intended only to authenticate the source. Private values must never be disclosed beyond the machine on which they were created.

5. Public encryption keys and certificates must be retrievable directly from servers without requiring secured channels; however, the fundamental security of identification credentials and public values bound to those credentials must be a function of certificate authorities and/or webs of trust.

6. Error checking must be at the enhanced paranoid level, as network terrorists may be able to craft errored packets that consume excessive cycles with needless result. While this document includes an informal vulnerability analysis and error protection paradigm, a formal model based on communicating finite-state machine analysis remains to be developed.

Unlike the Secure Shell security model, where the client must be securely authenticated to the server, in NTP the server must be securely authenticated to the client. In ssh each different interface address can be bound to a different name, as returned by a reverse-DNS query. In this design separate public/private key pairs may be required for each interface address with a distinct name. A perceived advantage of this design is that the security compartment can be different for each interface. This allows a firewall, for instance, to require some interfaces to proventicate the client and others not.

However, the NTP security model specifically assumes that access control is performed by means external to the protocol and that all time values and cryptographic values are public, so there is no need to associate each interface with different cryptographic values. To do so would create the possibility of a two-faced clock, which is ordinarily considered a Byzantine hazard. In other words, there is one set of private secrets for the host, not one for each interface. In the NTP design the host name, as returned by the Unix gethostname() library function, represents all interface addresses. Since at least in some host configurations the host name may not be identifiable in a DNS query, the name must be either configured in advance or obtained directly from the server using the Autokey protocol.

3. Approach

The Autokey protocol described in this document is designed to meet the following objectives. Again, in-depth discussions on these objectives is in the web briefings and will not be elaborated in this document. Note that here and elsewhere in this document mention of broadcast mode means multicast mode as well, with exceptions noted in the NTP software documentation.

1. It must interoperate with the existing NTP architecture model and protocol design. In particular, it must support the symmetric key scheme described in [11]. As a practical matter, the reference implementation must use the same internal key management system, including the use of 32-bit key IDs and existing mechanisms to store, activate and revoke keys.

2. It must provide for the independent collection of cryptographic values and time values. A NTP packet is accepted for processing only when the required cryptographic values have been obtained and verified and the NTP header has passed all sanity checks.

3. It must not significantly degrade the potential accuracy of the NTP synchronization algorithms. In particular, it must not make unreasonable demands on the network or host processor and memory resources.

4. It must be resistant to cryptographic attacks, specifically those identified in the security model above. In particular, it must be tolerant of operational or implementation variances, such as packet loss or misorder, or suboptimal configurations.

5. It must build on a widely available suite of cryptographic algorithms, yet be independent of the particular choice. In particular, it must not require data encryption other than incidental to signature and cookie encryption operations.

6. It must function in all the modes supported by NTP, including server, symmetric and broadcast modes.

7. It must not require intricate per-client or per-server configuration other than the availability of the required cryptographic keys and certificates.

8. The reference implementation must contain provisions to generate cryptographic key files specific to each client and server.

4. Autokey Cryptography

Autokey public key cryptography is based on the PKI algorithms commonly used in the Secure Shell and Secure Sockets Layer applications. As in these applications Autokey uses keyed message digests to detect packet modification, digital signatures to verify the source and public key algorithms to encrypt cookies. What makes Autokey cryptography unique is the way in which these algorithms are used to deflect intruder attacks while maintaining the integrity and accuracy of the time synchronization function.

NTPv3 and NTPv4 symmetric key cryptography use keyed-MD5 message digests with a 128-bit private key and 32-bit key ID. In order to retain backward compatibility with NTPv3, the NTPv4 key ID space is partitioned in two subspaces at a pivot point of 65536. Symmetric key IDs have values less than the pivot and indefinite lifetime. Autokey key IDs have pseudo-random values equal to or greater than the pivot and are expunged immediately after use. Both symmetric key and public key cryptography authenticate as shown below. The server looks up the key associated with the key ID and calculates the message digest from the NTP header and extension fields together with the key value. The key ID and digest form the message authentication code (MAC) included with the message. The client does the same computation using its local copy of the key

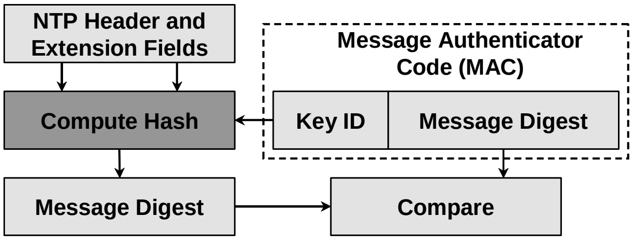
NTP Header and Extension Fields
Message Authenticator Code (MAC)
Compute Hash
Key ID
Message Digest
Message Digest
Compare

Figure 1. Receiving Messages

and compares the result with the digest in the MAC. If the values agree, the message is assumed authentic.

There are three Autokey protocol variants corresponding to each of the three NTP modes: server, symmetric and broadcast. All three variants make use of specially contrived session keys, called autokeys, and a precomputed pseudo-random sequence of autokeys with the key IDs saved in a key list. As in the original NTPv3 authentication scheme, the Autokey protocol operates separately for each association, so there may be several autokey sequences operating independently at the same time.

An autokey is computed from four fields in network byte order as shown below:

Source Address	Dest Address	Key ID	Cookie

Figure 2. NTPv4 Autokey

The four values are hashed by the MD5 message digest algorithm to produce the 128-bit autokey value, which in the reference implementation is stored along with the key ID in a cache used for symmetric keys as well as autokeys. Keys are retrieved from the cache by key ID using hash tables and a fast lookup algorithm.

For use with IPv4, the Source IP and Dest IP fields contain 32 bits; for use with IPv6, these fields contain 128 bits. In either case the Key ID and Cookie fields contain 32 bits. Thus, an IPv4 autokey has four 32-bit words, while an IPv6 autokey has ten 32-bit words. The source and destination IP addresses and key ID are public values visible in the packet, while the cookie can be a public value or shared private value, depending on the mode.

The NTP packet format has been augmented to include one or more extension fields piggybacked between the original NTP header and the message authenticator code (MAC) at the end of the packet. For packets without extension fields, the cookie is a shared private value conveyed in encrypted form. For packets with extension fields, the cookie has a default public value of zero, since these packets can be validated independently using digital signatures.

There are some scenarios where the use of endpoint IP addresses may be difficult or impossible. These include configurations where network address translation (NAT) devices are in use or when addresses are changed during an association lifetime due to mobility constraints. For Autokey, the only restriction is that the address fields visible in the transmitted packet must be the same as those used to construct the autokey sequence and key list and that these fields be the same as those visible in the received packet.

Provisions are included in the reference implementation to handle cases when these addresses change, as possible in mobile IP. For scenarios where the endpoint IP addresses are not available, an optional public identification value could be used instead of the addresses. Examples include the Interplanetary Internet, where bundles are identified by name rather than address. Specific provisions are for further study.

The figure below shows how the autokey list and autokey values are computed. The key list consists of a sequence of key IDs starting with a random 32-bit nonce (autokey seed) equal to or greater than the pivot as the first key ID. The first autokey is computed as above using the given cookie and the first 32 bits of the result in network byte order become the next key ID. Operations continue to generate the entire list. It may happen that a newly generated key ID is less than the pivot or collides with another one already generated (birthday event). When this happens, which occurs only rarely, the key list is terminated at that point. The lifetime of each key is set to expire one poll interval after its scheduled use. In the reference implementation, the list is terminated when the maximum key lifetime is about one hour, so for poll intervals above one hour a new key list containing only a single entry is regenerated for every poll.

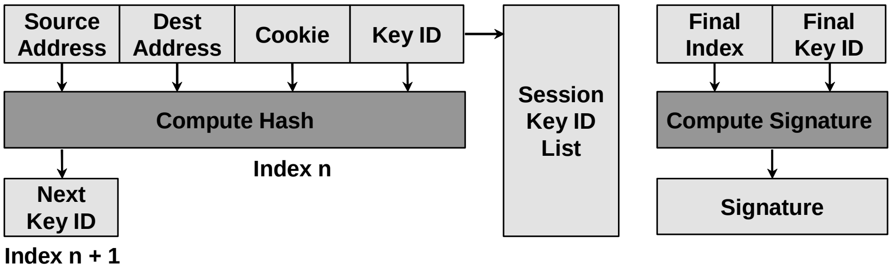

Figure 3. Constructing Key List

The index of the last key ID in the list is saved along with the next key ID for that entry, collectively called the autokey values. The autokey values are then signed. The list is used in reverse order as in the figure below, so that the first autokey used is the last one generated. The Autokey protocol includes a message to retrieve the autokey values and signature, so that subsequent packets can be validated using one or more hashes that eventually match the last key ID (valid) or exceed the index (invalid). This is called the autokey test in the following and is done for every packet, including those with and without extension fields. In the reference implementation the most recent key ID received is saved for comparison with the first 32 bits in network byte order of the next following key value. This minimizes the number of hash operations in case a packet is lost.

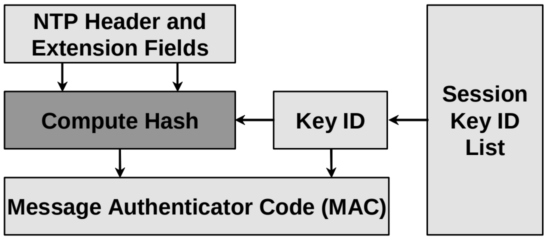
NTP Header and Extension Fields
Compute Hash
Key ID
Session Key ID List
Message Authenticator Code (MAC)

Figure 4. Transmitting Messages

5. Autokey Operations

The Autokey protocol has three variations, called dances, corresponding to the NTP server, symmetric and broadcast modes. The server dance was suggested by Steve Kent over lunch some time ago, but considerably modified since that meal. The server keeps no state for each client, but uses a fast algorithm and a 32-bit random private value (server seed) to regenerate the cookie upon arrival of a client packet. The cookie is calculated as the first 32 bits of the autokey computed from the client and server addresses, a key ID of zero and the server seed as cookie. The cookie is used for the actual autokey calculation by both the client and server and is thus specific to each client separately.

In previous Autokey versions the cookie was transmitted in clear on the assumption it was not useful to a wiretapper other than to launch an ineffective replay attack. However, a middleman could intercept the cookie and manufacture bogus messages acceptable to the client. In order to reduce the risk of such an attack, the Autokey Version 2 server encrypts the cookie using a public key supplied by the client. While requiring additional processor resources for the encryption, this makes it effectively impossible to spoof a cookie or masquerade as the server.

[Note in passing. In an attempt to avoid the use of overt encryption operations, an experimental scheme used a Diffie-Hellman agreed key as a stream cipher to encrypt the cookie. However, not only was the protocol extremely awkward, but the processing time to execute the agreement, encrypt the key and sign the result was horrifically expensive - 15 seconds in a vintage Sun IPC. This scheme was quickly dropped in favor of generic public key encryption.]

The server dance uses the cookie and each key ID on the key list in turn to retrieve the autokey and generate the MAC in the NTP packet. The server uses the same values to generate the message digest and verifies it matches the MAC in the packet. It then generates the MAC for the response using the same values, but with the client and server addresses exchanged. The client generates the message digest and verifies it matches the MAC in the packet. In order to deflect old replays, the client verifies the key ID matches the last one sent. In this mode the sequential structure of the key list is not exploited, but doing it this way simplifies and regularizes the implementation while making it nearly impossible for an intruder to guess the next key ID.

In broadcast dance clients normally do not send packets to the server, except when first starting up to verify credentials and calibrate the propagation delay. At the same time the client runs the broadcast dance to obtain the autokey values. The dance requires the association ID of the particular server association, since there can be more than one operating in the same server. For this purpose, the server packet includes the association ID in every response message sent and, when sending the first packet after generating a new key list, it sends the autokey values as well. After obtaining and verifying the autokey values, the client verifies further server packets using the autokey sequence.

The symmetric dance is similar to the server dance and keeps only a small amount of state between the arrival of a packet and departure of the reply. The key list for each direction is generated separately by each peer and used independently, but each is generated with the same cookie. The cookie is conveyed in a way similar to the server dance, except that the cookie is a random value. There exists a possible race condition where each peer sends a cookie request message before receiving the cookie response from the other peer. In this case, each peer winds up with two values, one it generated and one the other peer generated. The ambiguity is resolved simply by computing the working cookie as the EXOR of the two values.

Autokey choreography includes one or more exchanges, each with a specific purpose, that must be completed in order. The client obtains the server host name, digest/signature scheme and identity scheme in the parameter exchange. It recursively obtains and verifies certificates on the trail leading to a trusted certificate in the certificate exchange and verifies the server identity in the identity exchange. In the values exchange the client obtains the cookie and autokey values, depending on the particular dance. Finally, the client presents its self-signed certificate to the server for signature in the sign exchange.

The ultimate security of Autokey is based on digitally signed certificates and a certificate infrastructure compatible with [1] and [4]. The Autokey protocol builds the certificate trail from the primary servers, which presumably have trusted self-signed certificates, recursively by stratum. Each stratum $n + 2$ server obtains the certificate of a stratum n server, presumably signed by a stratum $n - 1$ server, and then the stratum $n + 1$ server presents its own self-signed certificate for signature by the stratum n server. As the NTP subnet forms from the primary servers at the root outward to the leaves, each server accumulates non-duplicative certificates for all associations and for all trails. In typical NTP subnets, this results in a good deal of useful redundancy and cross checking and making it even harder for a terrorist to subvert.

In order to prevent masquerade, it is necessary for the stratum n server to prove identity to the stratum $n + 1$ server when signing its certificate. In many applications a number of servers share a single security compartment, so it is only necessary that each server verifies identity to the group. Although no specific identity scheme is specified in this document, Appendix E describes a number of them based on cryptographic challenge-response algorithms. The reference implementation includes all of them with provision to add more if required.

Once the certificates and identity have been validated, subsequent packets are validated by digital signatures or autokey sequences. These packets are presumed to contain valid time values; however, unless the system clock has already been set by some other proventic means, it is not known whether these values actually represent a truechime or falsetick source. As the protocol evolves, the NTP associations continue to accumulate time values until a majority clique is available to synchronize the system clock. At this point the NTP intersection algorithm culls the falsetickers from the population and the remaining truechimers are allowed to discipline the clock.

The time values for truechimer sources form a proventic partial ordering relative to the applicable signature timestamps. This raises the interesting issue of how to mitigate between the timestamps of different associations. It might happen, for instance, that the timestamp of some Autokey message is ahead of the system clock by some presumably small amount. For this reason, timestamp comparisons between different associations and between associations and the system clock are avoided, except in the NTP intersection and clustering algorithms and when determining whether a certificate has expired.

Once the Autokey values have been instantiated, the dances are normally dormant. In all except the broadcast dance, packets are normally sent without extension fields, unless the packet is the first one sent after generating a new key list or unless the client has requested the cookie or autokey values. If for some reason the client clock is stepped, rather than slewed, all cryptographic and time values for all associations are purged and the dances in all associations restarted from scratch. This insures that stale values never propagate beyond a clock step. At intervals of about one day the reference implementation purges all associations, refreshes all signatures, garbage collects expired certificates and refreshes the server seed.

6. Public Key Signatures and Timestamps

While public key signatures provide strong protection against misrepresentation of source, computing them is expensive. This invites the opportunity for an intruder to clog the client or server by replaying old messages or to originate bogus messages. A client receiving such messages might be forced to verify what turns out to be an invalid signature and consume significant processor resources.

In order to foil such attacks, every signed extension field carries a timestamp in the form of the NTP seconds at the signature epoch. The signature spans the entire extension field including the timestamp. If the Autokey protocol has verified a proventic source and the NTP algorithms have validated the time values, the system clock can be synchronized and signatures will then carry a nonzero (valid) timestamp. Otherwise the system clock is unsynchronized and signatures carry a zero (invalid) timestamp. The protocol detects and discards replayed extension fields with old or duplicate timestamps, as well as fabricated extension fields with bogus timestamps, before any values are used or signatures verified.

There are three signature types currently defined:

1. Cookie signature/timestamp: Each association has a cookie for use when generating a key list. The cookie value is determined along with the cookie signature and timestamp upon arrival of a cookie request message. The values are returned in a a cookie response message.

2. Autokey signature/timestamp: Each association has a key list for generating the autokey sequence. The autokey values are determined along with the autokey signature and timestamp when a new key list is generated, which occurs about once per hour in the reference implementation. The values are returned in a autokey response message.

3. Public values signature/timestamp: All public key, certificate and leap second table values are signed at the time of generation, which occurs when the system clock is first synchronized to a proventic source, when the values have changed and about once per day after that, even if these values have not changed. During protocol operations, each of these values and associated signatures and timestamps are returned in the associated request or response message. While there are in fact several public value signatures, depending on the number of entries on the certificate list, the values are all signed at the same time, so there is only one public value timestamp.

The most recent timestamp received of each type is saved for comparison. Once a valid signature with valid timestamp has been received, messages with invalid timestamps or earlier valid timestamps of the same type are discarded before the signature is verified. For signed messages this deflects replays that otherwise might consume significant processor resources; for other messages the Autokey protocol deflects message modification or replay by a wiretapper, but not necessarily by a middleman. In addition, the NTP protocol itself is inherently resistant to replays and consumes only minimal processor resources.

All cryptographic values used by the protocol are time sensitive and are regularly refreshed. In particular, files containing cryptographic basis values used by signature and encryption algorithms are regenerated from time to time. It is the intent that file regenerations occur without specific advance warning and without requiring prior distribution of the file contents. While cryptographic data files are not specifically signed, every file is associated with a filestamp in the form of the NTP seconds at the creation epoch. It is not the intent in this document to specify file formats or names or encoding rules; however, whatever conventions are used must support a NTP filestamp in one form or another. Additional details specific to the reference implementation are in Appendix B.

Filestamps and timestamps can be compared in any combination and use the same conventions. It is necessary to compare them from time to time to determine which are earlier or later. Since these quantities have a granularity only to the second, such comparisons are ambiguous if the values are the same. Thus, the ambiguity must be resolved for each comparison operation as described in Appendix C.

It is important that filestamps be proventic data; thus, they cannot be produced unless the producer has been synchronized to a proventic source. As such, the filestamps throughout the NTP subnet represent a partial ordering of all creation epochs and serve as means to expunge old data and insure new data are consistent. As the data are forwarded from server to client, the filestamps are preserved, including those for certificate and leap seconds files. Packets with older filestamps are discarded before spending cycles to verify the signature.

7. Autokey Protocol Overview

This section presents an overview of the three dances: server, broadcast and symmetric. Each dance is designed to be non intrusive and to require no additional packets other than for regular NTP operations. The NTP and Autokey protocols operate independently and simultaneously and use the same packets. When the preliminary dance exchanges are complete, subsequent packets are validated by the autokey sequence and thus considered proventic as well. Autokey assumes clients poll servers at a relatively low rate, such as once per minute or slower. In particular, it is assumed that a request sent at one poll opportunity will normally result in a response before the next poll opportunity.

The Autokey protocol data unit is the extension field, one or more of which can be piggybacked in the NTP packet. An extension field contains either a request with optional data or a response with data. To avoid deadlocks, any number of responses can be included in a packet, but only one request. A response is generated for every request, even if the requestor is not synchronized to a proventic source, but contain meaningful data only if the responder is synchronized to a proventic source. Some requests and most responses carry timestamped signatures. The signature covers the entire extension field, including the timestamp and filestamp, where applicable. Only if the packet passes all extension field tests are cycles spent to verify the signature.

All dances begin with the parameter exchange where the client obtains the server host name and status word specifying the digest/signature scheme it will use and the identity schemes it supports. The dance continues with the certificate exchange where the client obtains and verifies the certificates along the trail to a trusted, self-signed certificate usually, but not necessarily, provided by a primary (stratum 1) server. Primary servers are by design proventic with trusted, self-signed certificates.

However, the certificate trail is not sufficient protection against middleman attacks unless an identity scheme such as described in Appendix E or proof-of-possession scheme in [14] is available.

While the protocol for a generic challenge/response scheme is defined in this document, the choice of one or another required or optional identification schemes is yet to be determined. If all certificate signatures along the trail are verified and the server identity is confirmed, the server is declared proventic. Once declared proventic, the client verifies packets using digital signatures and/or the autokey sequence.

Once synchronized to a proventic source, the client continues with the sign exchange where the server acting as CA signs the client certificate. The CA interprets the certificate as a X.509v3 certificate request, but verifies the signature if it is self-signed. The CA extracts the subject, issuer, extension fields and public key, then builds a new certificate with these data along with its own serial number and begin and end times, then signs it using its own public key. The client uses the signed certificate in its own role as CA for dependent clients.

In the server dance the client presents its public key and requests the server to generate and return a cookie encrypted with this key. The server constructs the cookie as described above and encrypts it using this key. The client decrypts the cookie for use in generating the key list. A similar dance is used in symmetric mode, where one peer acts as the client and the other the server. In case of overlapping messages, each peer generates a cookie and the agreed common value is computed as the EXOR of the two cookies.

The cookie is used to generate the key list and autokey values in all dances. In the server dance there is no need to provide these values to the server, so once the cookie has been obtained the client can generate the key list and validate succeeding packets directly. In other dances the client requests the autokey values from the server or, in some modes, the server provides them as each new key list is generated. Once these values have been received, the client validates succeeding packets using the autokey sequence as described previously.

A final exchange occurs when the server has the leap seconds table, as indicated in the host status word. If so, the client requests the table and compares the filestamp with its own leap seconds table filestamp, if available. If the server table is newer than the client table, the client replaces its table with the server table. The client, acting as server, can now provide the most recent table to any of its dependent clients. In symmetric mode, this results in both peers having the newest table.

8. Autokey State Machine

This section describes the formal model of the Autokey state machine, its state variables and the state transition functions.

8.1 Status Word

Each server and client operating also as a server implements a host status word, while each client implements a server status word for each server. Both words have the format and content shown below.

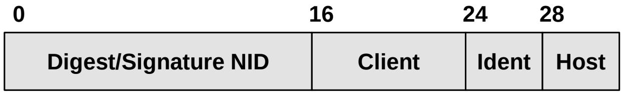

Figure 5. Status Word

The low order 16 bits of the status word define the state of the Autokey protocol, while the high order 16 bits specify the message digest/signature encryption scheme. Bits 24-31 of the status word are reserved for server use, while bits 16-23 are reserved for client use. There are four additional bits implemented separately.

The host status word is included in the ASSOC request and response messages. The client copies this word to the server status word and then lights additional status bits as the dance proceeds. Once lit, these bits never come dark unless a general reset occurs and the protocol is restarted from the beginning. The status bits are defined as follows:

`ENB` (31)

Lit if the server implements the Autokey protocol and is prepared to dance. Dim would be very strange.

`LPF` (30)

Lit if the server has loaded a valid leap seconds file. This bit can be either lit or dim.

`IDN` (24-27)

These four bits select which identity scheme is in use. While specific coding for various schemes is yet to be determined, the schemes available in the reference implementation and described in Appendix E include the following.

0x0 Trusted Certificate (TC) Scheme (default)
0x1 Private Certificate (PC) Scheme
0x2 Schnorr aka Identify-Friendly-or-Foe (IFF) Scheme
0x4 Guillard-Quisquater (GC) Scheme
0x8 Mu-Varadharajan (MV) Scheme

The PC scheme is exclusive of any other scheme. Otherwise, the IFF, GQ and MV bits can be lit in any combination.

The server status bits are defined as follows:

`VAL` 0x0100

Lit when the server certificate and public key are validated.

`IFF` 0x0200

Lit when the server identity credentials are confirmed by one of several schemes described later.

`PRV` 0x0400

Lit when the server signature is verified using the public key and identity credentials. Also called the proventic bit elsewhere in this document. When lit, signed values in subsequent messages are presumed proventic, but not necessarily time-synchronized.

`CKY` 0x0800

Lit when the cookie is received and validated. When lit, key lists can be generated.

AUT 0x1000

Lit when the autokey values are received and validated. When lit, clients can validate packets without extension fields according to the autokey sequence.

SGN 0x2000

Lit when the host certificate is signed by the server.

LPT 0x4000

Lit when the leap seconds table is received and validated.

There are four additional status bits LST, LBK, DUP and SYN not included in the status word. All except SYN are association properties, while SYN is a host property. These bits may be lit or dim as the protocol proceeds; all except LST are active whether or not the protocol is running. LST is lit when the key list is regenerated and signed and comes dim after the autokey values have been transmitted. This is necessary to avoid livelock under some conditions. SYN is lit when the client has synchronized to a proventic source and never dim after that. There are two error bits: LBK indicates the received packet does not match the last one sent and DUP indicates a duplicate packet. These bits, which are described in Appendix C, are lit if the corresponding error has occurred for the current packet and dim otherwise.

8.2 Host State Variables

Host Name

The name of the host returned by the Unix gethostname() library function. The name must agree with the subject name in the host certificate.

Host Status Word

This word is initialized when the host first starts up. The format is described above.

Host Key

The RSA public/private key used to encrypt/decrypt cookies. This is also the default sign key.

Sign Key

The RSA or DSA public/private key used to encrypt/decrypt signatures when the host key is not used for this purpose.

Sign Digest

The message digest algorithm used to compute the signature before encryption.

IFF Parameters

The parameters used in the IFF identity scheme described in Appendix E.

GQ Parameters

The parameters used in the GQ identity scheme described in Appendix E.

MV Parameters

The parameters used in the MV identity scheme described in Appendix E.

Server Seed

The private value hashed with the IP addresses to construct the cookie.

Certificate Information Structure (CIS)

Certificates are used to construct certificate information structures (CIS) which are stored on the certificate list. The structure includes certain information fields from an X.509v3 certificate, together with the certificate itself encoded in ASN.1 syntax. Each structure carries the public value timestamp and the filestamp of the certificate file where it was generated. Elsewhere in this document the CIS will not be distinguished from the certificate unless noted otherwise.

A flags field in the CIS determines the status of the certificate. The field is encoded as follows:

`SIGN` 0x01

The certificate signature has been verified. If the certificate is self-signed and verified using the contained public key, this bit will be lit when the CIS is constructed.

`TRST` 0x02

The certificate has been signed by a trusted issuer. If the certificate is self-signed and contains “trustRoot” in the Extended Key Usage field, this bit will be lit when the CIS is constructed.

`PRIV` 0x04

The certificate is private and not to be revealed. If the certificate is self-signed and contains “Private” in the Extended Key Usage field, this bit will be lit when the CIS is constructed.

`ERRR` 0x80

The certificate is defective and not to be used in any way.

Certificate List

CIS structures are stored on the certificate list in order of arrival, with the most recently received CIS placed first on the list. The list is initialized with the CIS for the host certificate, which is read from the certificate file. Additional CIS entries are pushed on the list as certificates are obtained from the servers during the certificate exchange. CIS entries are discarded if overtaken by newer ones or expire due to old age.

Host Certificate

The self-signed X.509v3 certificate for the host. The subject and issuer fields consist of the host name, while the message digest/signature encryption scheme consists of the sign key and message digest defined above. Optional information used in the identity schemes is carried in X.509v3 extension fields compatible with [4].

Public Key Values

The public encryption key for the COOKIE request, which consists of the public value of the host key. It carries the public values timestamp and the filestamp of the host key file.

Leapseconds Table Values

The NIST leap seconds table from the NIST leap seconds file. It carries the public values timestamp and the filestamp of the leap seconds file.

8.3 Client State Variables (all modes)

Association ID

The association ID used in responses. It is assigned when the association is mobilized.

Server Association ID

The server association ID used in requests. It is initialized from the first nonzero association ID field in a response.

Server Subject Name

The server host name determined in the parameter exchange.

Server Issuer Name

The host name signing the certificate. It is extracted from the current server certificate upon arrival and used to request the next item on the certificate trail.

Server Status Word

The host status word of the server determined in the parameter exchange.

Server Public Key

The public key used to decrypt signatures. It is extracted from the first certificate received, which by design is the server host certificate.

Server Message Digest

The digest/signature scheme determined in the parameter exchange.

Identification Challenge

A 512-bit nonce used in the identification exchange.

Group Key

A 512-bit secret group key used in the identification exchange. It identifies the cryptographic compartment shared by the server and client.

Receive Cookie Values

The cookie returned in a COOKIE response, together with its timestamp and filestamp.

Receive Autokey Values

The autokey values returned in an AUTO response, together with its timestamp and filestamp.

Receive Leap second Values

The leap second table returned by a LEAP response, together with its timestamp and filestamp.

8.4 Server State Variables (broadcast and symmetric modes)

Send Cookie Values

The cookie encryption values, signature and timestamps.

Send Autokey Values

The autokey values, signature and timestamps.

Key List

A sequence of key IDs starting with the autokey seed and each pointing to the next. It is computed, timestamped and signed at the next poll opportunity when the key list becomes empty.

Current Key Number

The index of the entry on the Key List to be used at the next poll opportunity.

8.5 Autokey Messages

There are currently eight Autokey requests and eight corresponding responses. A description of these messages is given below; the detailed field formats are described in Appendix A.

8.5.1 Association Message (ASSOC)

The Association message is used in the parameter exchange to obtain the host name and related values. The request contains the host status word in the filestamp field. The response contains the status word in the filestamp field and in addition the host name as the unterminated string returned by the Unix gethostname() library function. While minimum and maximum host name lengths remain to be established, the reference implementation uses the values 4 and 256, respectively. The remaining fields are defined previously in this document.

If the server response is acceptable and both server and client share the same identity scheme, `ENB` is lit. When the PC identity scheme is in use, the ASSOC response lights `VAL`, `IFF` and `SIG`, since the IFF exchange is complete at this point.

8.5.2 Certificate Message (CERT)

The Certificate message is used in the certificate exchange to obtain a certificate and related values by subject name. The request contains the subject name. For the purposes of interoperability with older Autokey versions, if only the first two words are sent, the request is for the host certificate. The response contains the certificate encoded in X.509 format with ASN.1 syntax as described in Appendix G.

If the subject name in the response does not match the issuer name, the exchange continues with the issuer name replacing the subject name in the request. The exchange continues until either the subject name matches the issuer name, indicating a self-signed certificate, or the `trst` bit is set in the CIS, indicating a trusted certificate. If a trusted certificate is found, the client stops the exchange and lights `VAL.`If a self-signed certificate is found without encountering a trusted certificate, the protocol loops until either a new certificate is signed or timeout.

8.5.3 Cookie Message (COOKIE)

The Cookie message is used in server and symmetric modes to obtain the server cookie. The request contains the host public key encoded with ASN.1 syntax as described in Appendix G. The response contains the cookie encrypted by the public key in the request. The signature and timestamps are determined when the cookie is encrypted. If the response is valid, the client lights `CKY`.

8.5.4 Autokey Message (AUTO)

The Autokey message is used to obtain the autokey values. The request contains no value. The response contains two 32-bit words in network order. The first word is the final key ID, while the second is the index of the final key ID. The signature and timestamps are determined when the key list is generated. If the response is valid, the client lights `AUT`.

8.5.5 Leapseconds Table Message (LEAP)

The Leapseconds Table message is used to exchange leap seconds tables. The request and response messages have the same format, except that the `R` bit is dim in the request and lit in the response. Both the request and response contains the leap seconds table as parsed from the leap seconds file from NIST. If the client already has a copy of the leap seconds data, it uses the one with the latest filestamp and discards the other. If the response is valid, the client lights `LPT`.

8.5.6 Sign Message (SIGN)

The Sign message requests the server to sign and return a certificate presented in the request. The request contains the client certificate encoded in X.509 format with ASN.1 syntax as described in Appendix G. The response contains the client certificate signed by the server private key. If the certificate is valid when received by the client, it is linked in the certificate list and the client lights `SGN`.

8.5.7 Identity Messages (IFF, GQ, MV)

The request contains the client challenge, usually a 160- or 512-bit nonce. The response contains the result of the mathematical operation defined in Appendix E. The Response is encoded in ASN.1 syntax as described in Appendix G. The response signature and timestamp are determined when the response is sent. If the response is valid, the client lights `IFF`.

8.6 Protocol State Transitions

The protocol state machine is very simple but robust. The state is determined by the server status bits defined above. The state transitions of the three dances are shown below. The capitalized truth values represent the server status bits. All server bits are initialized dark and light up upon the arrival of a specific response message, as detailed above.

When the system clock is first set and about once per day after that, or when the system clock is stepped, the server seed is refreshed, signatures and timestamps updated and the protocol restarted in all associations. When the server seed is refreshed or a new certificate or leap second table is received, the public values timestamp is reset to the current time and all signatures are recomputed.

8.6.1 Server Dance

The server dance begins when the client sends an ASSOC request to the server. It ends when the first signature is verified and PRV is lit. Subsequent packets received without extension fields are validated by the autokey sequence. An optional LEAP exchange updates the leap seconds table. Note the order of the identity exchanges and that only the first one will be used if multiple schemes are available. Note also that the SIGN and LEAP requests are not issued until the client has synchronized to a proventic source.

```
   while (1) {
      wait_for_next_poll;
      make_NTP_header;
      if (response_ready)
         send_response;
      if (!ENB)                     /* parameters exchange */
         ASSOC_request;
      else if (!VAL)                /* certificate exchange */
         CERT_request(Host_Name);
      else if (IDN & GQ && !IFF)    /* GQ identity exchange */
         GQ_challenge;
      else if (IDN & IFF && !IFF)   /* IFF identity exchange */
         IFF_challenge;
      else if (!IFF)                /* TC identity exchange */
         CERT_request(Issuer_Name);
      else if (!CKY)                /* cookie exchange */
         COOKIE_request;
      else if (SYN && !SIG)         /* sign exchange */
         SIGN_request(Host_Certificate);
      else if (SYN && LPF & !LPT)   /* leapseconds exchange */
         LEAP_request;
}
```

When the PC identity scheme is in use, the ASSOC response lights VAL, IFF and SIG, the COOKIE response lights CKY and AUT and the first valid signature lights PRV.

8.6.2 Broadcast Dance

THe only difference between the broadcast and server dances is the inclusion of an autokey values exchange following the cookie exchange. The broadcast dance begins when the client receives the first broadcast packet, which includes an ASSOC response with association ID. The broadcast client uses the association ID to initiate a server dance in order to calibrate the propagation delay.

The dance ends when the first signature is verified and PRV is lit. Subsequent packets received without extension fields are validated by the autokey sequence. An optional LEAP exchange updates the leapseconds table. When the server generates a new key list, the server replaces the ASSOC response with an AUTO response in the first packet sent.

```
while (1) {
   wait_for_next_poll;
   make_NTP_header;
   if (response_ready)
      send_response;
   if (!ENB)                     /* parameters exchange */
      ASSOC_request;
   else if (!VAL)                /* certificate exchange */
      CERT_request(Host_Name);
   else if (IDN & GQ && !IFF)    /* GQ identity exchange */
      GQ_challenge;
   else if (IDN & IFF && !IFF)   /* IFF identity exchange */
      IFF_challenge;
   else if (!IFF)                /* TC identity exchange */
      CERT_request(Issuer_Name);
   else if (!CKY)                /* cookie exchange */
      COOKIE_request;
   else if (!AUT)                /* autokey values exchange */
      AUTO_request;
   else if (SYN &&! SIG)         /* sign exchange */
      SIGN_request(Host_Certificate);
   else if (SYN && LPF & !LPT)   /* leapseconds exchange */
      LEAP_request;
}
```

When the PC identity scheme is in use, the ASSOC response lights VAL, IFF and SIG, the COOKIE response lights CKY and AUT and the first valid signature lights PRV.

8.6.3 Symmetric Dance

The symmetric dance is intricately choreographed. It begins when the active peer sends an ASSOC request to the passive peer. The passive peer mobilizes an association and both peers step the same dance from the beginning. Until the active peer is synchronized to a proventic source (which could be the passive peer) and can sign messages, the passive peer loops waiting for the timestamp in the ASSOC response to light up. Until then, the active peer dances the server steps, but skips the sign, cookie and leapseconds exchanges.

```
while (1) {
   wait_for_next_poll;
   make_NTP_header;
   if (!ENB)                     /* parameters exchange */
      ASSOC_request;
   else if (!VAL)                /* certificate exchange */
      CERT_request(Host_Name);
   else if (IDN & GQ && !IFF)    /* GQ identity exchange */
      GQ_challenge;
   else if (IDN & IFF && !IFF)   /* IFF identity exchange */
      IFF_challenge;
      else if (!IFF)               /* TC identity exchange */
         CERT_request(Issuer_Name);
      else if (SYN && !SIG)        /* sign exchange */
         SIGN_request(Host_Certificate);
      else if (SYN && !CKY)        /* cookie exchange */
         COOKIE_request;
      else if (!LST)               /* autokey values response */
         AUTO_response;
      else if (!AUT)               /* autokey values exchange */
         AUTO_request;
      else if (SYN && LPF & !LPT)  /* leapseconds exchange */
         LEAP_request;
}
```

When the PC identity scheme is in use, the ASSOC response lights VAL, IFF and SIG, the COOKIE response lights CKY and AUT and the first valid signature lights PRV.

Once the active peer has synchronized to a proventic source, it includes timestamped signatures with its messages. The first thing it does after lighting timestamps is dance the sign exchange so that the passive peer can survive the default identity exchange, if necessary. This is pretty weird, since the passive peer will find the active certificate signed by its own public key.

The passive peer, which has been stalled waiting for the active timestamps to light up, now mates the dance. The initial value of the cookie is zero. If a COOKIE response has not been received by either peer, the next message sent is a COOKIE request. The recipient rolls a random cookie, lights CKY and returns the encrypted cookie. The recipient decrypts the cookie and lights CKY. It is not a protocol error if both peers happen to send a COOKIE request at the same time. In this case both peers will have two values, one generated by itself peer and the other received from the other peer. In such cases the working cookie is constructed as the EXOR of the two values.

At the next packet transmission opportunity, either peer generates a new key list and lights LST; however, there may already be an AUTO request queued for transmission and the rules say no more than one request in a packet. When available, either peer sends an AUTO response and dims LST. The recipient initializes the autokey values, dims LST and lights AUT. Subsequent packets received without extension fields are validated by the autokey sequence.

The above description assumes the active peer synchronizes to the passive peer, which itself is synchronized to some other source, such as a radio clock or another NTP server. In this case, the active peer is operating at a stratum level one greater than the passive peer and so the passive peer will not synchronize to it unless it loses its own sources and the active peer itself has another source.

9. Error Recovery

The Autokey protocol state machine includes provisions for various kinds of error conditions that can arise due to missing files, corrupted data, protocol violations and packet loss or misorder, not to mention hostile intrusion. This section describes how the protocol responds to reachability and timeout events which can occur due to such errors. Appendix C contains an extended discussion on error checking and timestamp validation.

A persistent NTP association is mobilized by an entry in the configuration file, while an ephemeral association is mobilized upon the arrival of a broadcast, manycast or symmetric active packet. A general reset re initializes all association variables to the initial state when first mobilized. In addition, if the association is ephemeral, the association is demobilized and all resources acquired are returned to the system.

Every NTP association has two variables which maintain the liveness state of the protocol, the 8-bit reachability register defined in [11] and the watchdog timer, which is new in NTPv4. At every poll interval the reachability register is shifted left, the low order bit is dimmed and the high order bit is lost. At the same time the watchdog counter is incremented by one. If an arriving packet passes all authentication and sanity checks, the rightmost bit of the reachability register is lit and the watchdog counter is set to zero. If any bit in the reachability register is lit, the server is reachable, otherwise it is unreachable.

When the first poll is sent by an association, the reachability register and watchdog counter are zero. If the watchdog counter reaches 16 before the server becomes reachable, a general reset occurs. This resets the protocol and clears any acquired state before trying again. If the server was once reachable and then becomes unreachable, a general reset occurs. In addition, if the watchdog counter reaches 16 and the association is persistent, the poll interval is doubled. This reduces the network load for packets that are unlikely to elicit a response.

At each state in the protocol the client expects a particular response from the server. A request is included in the NTP packet sent at each poll interval until a valid response is received or a general reset occurs, in which case the protocol restarts from the beginning. A general reset also occurs for an association when an unrecoverable protocol error occurs. A general reset occurs for all associations when the system clock is first synchronized or the clock is stepped or when the server seed is refreshed.

There are special cases designed to quickly respond to broken associations, such as when a server restarts or refreshes keys. Since the client cookie is invalidated, the server rejects the next client request and returns a crypto-NAK packet. Since the crypto-NAK has no MAC, the problem for the client is to determine whether it is legitimate or the result of intruder mischief. In order to reduce the vulnerability in such cases, the crypto-NAK, as well as all responses, is believed only if the result of a previous packet sent by the client and not a replay, as confirmed by the LBK and DUP status bits described above. While this defense can be easily circumvented by a middleman, it does deflect other kinds of intruder warfare.

There are a number of situations where some event happens that causes the remaining autokeys on the key list to become invalid. When one of these situations happens, the key list and associated autokeys in the key cache are purged. A new key list, signature and timestamp are generated when the next NTP message is sent, assuming there is one. Following is a list of these situations.

4. When the cookie value changes for any reason.

5. When a client switches from server mode to broadcast mode. There is no further need for the key list, since the client will not transmit again.

6. When the poll interval is changed. In this case the calculated expiration times for the keys become invalid.

7. If a problem is detected when an entry is fetched from the key list. This could happen if the key was marked non-trusted or timed out, either of which implies a software bug.

10. References

1. Adams, C., S. Farrell. Internet X.509 public key infrastructure certificate management protocols. Network Working Group Request for Comments RFC-2510, Entrust Technologies, March 1999, 30 pp.

2. Bassham, L., W. Polk and R. Housley, “Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation Lists (CRL) Profile,” RFC-3279, April 2002.

3. Guillou, L.C., and J.-J. Quisquatar. A “paradoxical” identity-based signature scheme resulting from zero-knowledge. Proc. *CRYPTO 88 Advanced in Cryptology*, Springer-Verlag, 1990, 216-231.

4. Housley, R., et al. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Network Working Group Request for Comments RFC-3280, RSA Laboratories, April 2002, 129 pp.

5. Karn, P., and W. Simpson, “Photuris: Session-key Management Protocol”, RFC-2522, March 1999.

6. Kent, S., R. Atkinson, “IP Authentication Header,” RFC-2402, November 1998.

7. Kent, S., and R. Atkinson, “IP Encapsulating Security Payload (ESP),” RFC-2406, November 1998.

8. Maughan, D., M. Schertler, M. Schneider, and J. Turner, “Internet Security Association and Key Management Protocol (ISAKMP),” RFC-2408, November 1998.

9. Mills, D.L. Public key cryptography for the Network Time Protocol. Electrical Engineering Report 00-5-1, University of Delaware, May 2000. 23 pp.

10. Mills, D.L. Proposed authentication enhancements for the Network Time Protocol version 4. Electrical Engineering Report 96-10-3, University of Delaware, October 1996, 36 pp.

11. Mills, D.L., “Network Time Protocol (Version 3) Specification, Implementation and Analysis,” RFC-1305, March 1992.

12. Mu, Y., and V. Varadharajan. Robust and secure broadcasting. *Proc. INDOCRYPT 2001, LNCS 2247*, Springer Verlag, 2001, 223-231.

13. Orman, H., “The OAKLEY Key Determination Protocol,” RFC-2412, November 1998.

14. Prafullchandra, H., and J. Schaad. Diffie-Hellman proof-of-possession algorithms. Network Working Group Request for Comments RFC-2875, Critical Path, Inc., July 2000, 23 pp.

15. Schnorr, C.P. Efficient signature generation for smart cards. *J. Cryptology 4*, 3 (1991), 161-174.

16. Stinson, D.R. *Cryptography - Theory and Practice*. CRC Press, Boca Raton, FA, 1995, ISBN 0-8493-8521-0.

A. Packet Formats

The NTPv4 packet consists of a number of fields made up of 32-bit (4 octet) words in network byte order. The packet consists of three components, the header, one or more optional extension fields and an optional authenticator.

A.1 Header Field Format

The header format is shown below, where the size of some fields is shown in bits if not the default 32 bits.

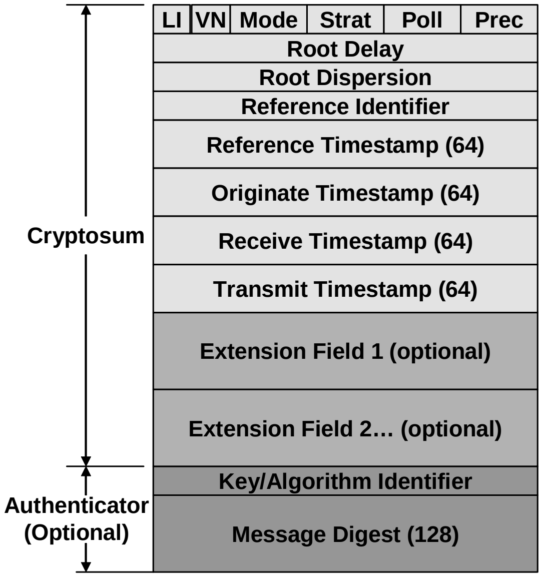

Figure 6. NTP Header Format

The NTP header extends from the beginning of the packet to the end of the Transmit Timestamp field. The format and interpretation of the header fields are backwards compatible with the NTPv3 header fields as described in [11].

A non-authenticated NTP packet includes only the NTP header, while an authenticated one contains in addition an authenticator which takes the form of a message authentication code (MAC). The MAC consisting of the Key ID and Message Digest fields.The format and interpretation of the NTPv4 MAC is described in [11] when using the Digital Encryption Standard (DES) algorithm operating in Cipher-Block Chaining (CBC) node. This algorithm and mode of operation is no longer supported in NTPv4. The preferred replacement in both NTPv3 and NTPv4 is the Message Digest 5 (MD5) algorithm, which is included in both reference implementations. For MD5 the Message Digest field is 4 words (8 octets) and the Key ID field is 1 word (4 octets).

A.2 Extension Field Format

In NTPv4 one or more extension fields can be inserted after the NTP header and before the MAC, which is always present when an extension field is present. The extension fields can occur in any order; however, in some cases there is a preferred order which improves the protocol efficiency.

While previous versions of the Autokey protocol used several different extension field formats, in version 2 of the protocol only a single extension field format is used.

Each extension field contains a request or response message in the following format:

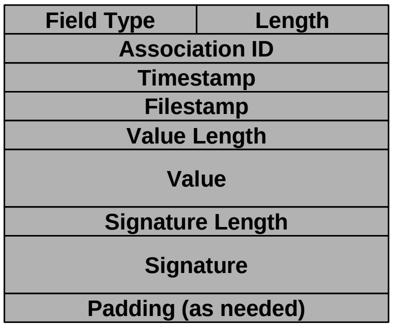

Figure 7. NTP Extension Field Format

Each extension field except the last is zero-padded to a word (4 octets) boundary, while the last is zero-padded to a doubleword (8 octets) boundary. The Length field covers the entire extension field, including the Length and Padding fields. While the minimum field length is 8 octets, a maximum field length remains to be established. The reference implementation discards any packet with a field length more than 1024 octets.

The presence of the MAC and extension fields in the packet is determined from the length of the remaining area after the header to the end of the packet. The parser initializes a pointer just after the header. If the length is not a multiple of 4, a format error has occurred and the packet is discarded. The following cases are possible based on the remaining length in words.

0
The packet is not authenticated.

4
The packet is an error report or crypto-NAK resulting from a previous packet that failed the message digest check. The 4 octets are presently unused and should be set to 0.

2, 3, 4
The packet is discarded with a format error.

5
The remainder of the packet is the MAC.

>5
One or more extension fields are present.

If an extension field is present, the parser examines the Length field. If the length is less than 4 or not a multiple of 4, a format error has occurred and the packet is discarded; otherwise, the parser increments the pointer by this value. The parser now uses the same rules as above to determine whether a MAC is present and/or another extension field. An additional implementation-dependent test is necessary to ensure the pointer does not stray outside the buffer space occupied by the packet.

In the Autokey Version 2 format, the Code field specifies the request or response operation, while the Version field is 2 for the current protocol version. There are two flag bits defined. Bit 0 is the response flag (R) and bit 1 is the error flag (E); the other six bits are presently unused and should be set to 0. The remaining fields will be described later.

In the most common protocol operations, a client sends a request to a server with an operation code specified in the Code field and lights the R bit. Ordinarily, the client dims the E bit as well, but may in future light it for some purpose. The Association ID field is set to the value previously received from the server or 0 otherwise. The server returns a response with the same operation code in the Code field and the R bit lit. The server can also light E bit in case of error. The Association ID field is set to the association ID sending the response as a handle for subsequent exchanges. If for some reason the association ID value in a request does not match the association ID of any mobilized association, the server returns the request with both the R and E bits lit. Note that it is not a protocol error to send an unsolicited response with no matching request.

In some cases not all fields may be present. For requests, until a client has synchronized to a proventic source, signatures are not valid. In such cases the Timestamp and Signature Length fields are 0 and the Signature field is empty. Responses are generated only when the responder has synchronized to a proventic source; otherwise, an error response message is sent. Some request and error response messages carry no value or signature fields, so in these messages only the first two words are present.

The Timestamp and Filestamp words carry the seconds field of an NTP timestamp. The Timestamp field establishes the signature epoch of the data field in the message, while the filestamp establishes the generation epoch of the file that ultimately produced the data that is signed. Since a signature and timestamp are valid only when the signing host is synchronized to a proventic source and a cryptographic data file can only be generated if a signature is possible, the response filestamp is always nonzero, except in the Association response message, where it contains the server status word.

B. Cryptographic Key and Certificate Management

This appendix describes how cryptographic keys and certificates are generated and managed in the NTPv4 reference implementation. These means are not intended to become part of any standard that may be evolved from this document, but to serve as an example of how these functions can be implemented and managed in a typical operational environment.

The ntp-keygen utility program in the NTP software library generates public/private key files, certificate files, identity parameter files and public/private identity key files. By default the modulus of all encryption and identity keys is 512 bits. All random cryptographic data are based on a pseudo-random number generator seeded in such a way that random values are exceedingly unlikely to repeat. The files are PEM encoded in printable ASCII format suitable for mailing as MIME objects.

Every file has a filestamp, which is a string of decimal digits representing the NTP seconds the file was created. The file name is formed from the concatenation of the host name, filestamp and constant strings, so files can be copied from one environment to another while preserving the original filestamp. The file header includes the file name and date and generation time in printable ASCII. The utility assumes the host is synchronized to a proventic source at the time of generation, so that filestamps are proventic data. This raises an interesting circularity issue that will not be further explored here.

The generated files are typically stored in NFS mounted file systems, with files containing private keys obscured to all but root. Symbolic links are installed from default file names assumed by the NTP daemon to the selected files. Since the files of successive generations and different hosts have unique names, there is no possibility of name collisions.

Public/private keys must be generated by the host to which they belong. OpenSSL public/private RSA and DSA keys are generated as an OpenSSL structure, which is then PEM encoded in ASN.1 syntax and written to the host key file. The host key must be RSA, since it is used to encrypt the cookie, as well as encrypt signatures by default. In principle, these files could be generated directly by OpenSSL utility programs, as long as the symbolic links are consistent. The optional sign key can be RSA or DSA, since it is used only to encrypt signatures.

Identity parameters must be generated by the ntp-keygen utility, since they have proprietary formats. Since these are private to the group, they are generated by one machine acting as a trusted authority and then distributed to all other members of the group by secure means. Public/private identity keys are generated by the host to which they belong along with certificates with the public identity key.

Certificates are usually, but not necessarily, generated by the host to which they belong. The ntp-keygen utility generates self-signed X.509v3 host certificate files with optional extension fields. Certificate requests are not used, since the certificate itself is used as a request to be signed. OpenSSL X.509v3 certificates are generated as an OpenSSL structure, which is then PEM encoded in ASN.1 syntax and written to the host certificate file. The string returned by the Unix gethostname() routine is used for both the subject and issuer fields. The serial number and begin time fields are derived from the filestamp; the end time is one year hence. The host certificate is signed by the sign key or host key by default.

An important design goal is to make cryptographic data refreshment as simple and intuitive as possible, so it can be driven by scripts on a periodic basis. When the ntp-keygen utility is run for the first time, it creates by default a RSA host key file and RSA-MD5 host certificate file and necessary symbolic links. After that, it creates a new certificate file and symbolic link using the existing host key. The program run with given options creates identity parameter files for one or both the IFF or GQ identity schemes. The parameter files must then be securely copied to all other group members and symbolic links installed from the default names to the installed files. In the GQ scheme the next and each subsequent time the ntp-keygen utility runs, it automatically creates or updates the private/public identity key file and certificate file using the existing identity parameters.

C. Autokey Error Checking

Exhaustive examination of possible vulnerabilities at the various processing steps of the NTPv3 protocol as specified in [11] have resulted in a revised list of packet sanity tests. There are 12 tests in the NTPv4 reference implementation, called TEST1 through TEST12, which are performed in a specific order designed to gain maximum diagnostic information while protecting against an accidental or malicious clogging attacks. These tests are described in detail in the NTP software documentation. Those relevant to the Autokey protocol are described in this appendix.

The sanity tests are classified in four tiers. The first tier deflects access control and message digest violations. The second, represented by the autokey sequence, deflects spoofed or replayed packets. The third, represented by timestamped digital signatures, binds cryptographic values to verifiable credentials. The fourth deflects packets with invalid NTP header fields or out of bounds time values. However, the tests in this last group do not directly affect cryptographic protocol vulnerability, so are beyond the scope of discussion here.

C.1 Packet Processing Rules

Every arriving NTP packet is checked enthusiastically for format, content and protocol errors. Some packet header fields are checked by the main NTP code path both before and after the Autokey protocol engine cranks. These include the NTP version number, overall packet length and extension field lengths. Extension fields may be no longer than 1024 octets in the reference implementation. Packets failing any of these checks are discarded immediately. Packets denied by the access control mechanism will be discarded later, but processing continues temporarily in order to gather further information useful for error recovery and reporting.

Next, the cookie and session key are determined and the MAC computed as described above. If the MAC fails to match the value included in the packet, the action depends on the mode and the type of packet. Packets failing the MAC check will be discarded later, but processing continues temporarily in order to gather further information useful for error recovery and reporting.

The NTP transmit and receive timestamps are in effect nonces, since an intruder cannot effectively guess either value in advance. To minimize the possibility that an intruder can guess the nonces, the low order unused bits in all timestamps are obscured with random values. If the transmit timestamp matches the transmit timestamp in the last packet received, the packet is a duplicate, so the DUP bit is lit. If the packet mode is not broadcast and the last transmit timestamp does not match the originate timestamp in the packet, either it was delivered out of order or an intruder has injected a rogue packet, so the LBK bit is lit. Packets with either the DUP or LBK bit lie be discarded later, but processing continues temporarily in order to gather further information useful for error recovery and reporting.

Further indicators of the server and client state are apparent from the transmit and receive timestamps of both the packet and the association. The quite intricate rules take into account these and the above error indicators They are designed to discriminate between legitimate cases where the server or client are in inconsistent states and recoverable, and when an intruder is trying to destabilize the protocol or force consumption of needless resources. The exact behavior is beyond the scope of discussion, but is clearly described in the source code documentation.

Next, the Autokey protocol engine is cranked and the dances evolve as described above. Some requests and all responses have value fields which carry timestamps and filestamps. When the server or client is synchronized to a proventic source, most requests and responses with value fields carry signatures with valid timestamps. When not synchronized to a proventic source, value fields carry an invalid (zero) timestamp and the signature field and signature length word are omitted.

The extension field parser checks that the Autokey version number, operation code and field length are valid. If the error bit is lit in a request, the request is discarded without response; if an error is discovered in processing the request, or if the responder is not synchronized to a proventic source, the response contains only the first two words of the request with the response and error bits lit. For messages with signatures, the parser requires that timestamps and filestamps are valid and not a replay, that the signature length matches the certificate public key length and only then verifies the signature. Errors are reported via the security logging facility.

All certificates must have correct ASN.1 encoding, supported digest/signature scheme and valid subject, issuer, public key and, for self-signed certificates, valid signature. While the begin and end times can be checked relative to the filestamp and each other, whether the certificate is valid relative to the actual time cannot be determined until the client is synchronized to a proventic source and the certificate is signed and verified by the server.

When the protocol starts the only response accepted is ASSOC with valid timestamp, after which the server status word must be nonzero. ASSOC responses are discarded if this word is nonzero. The only responses accepted after that and until the PRV bit is lit are CERT, IFF and GQ. Once identity is confirmed and IFF is lit, these responses are no longer accepted, but all other responses are accepted only if in response to a previously sent request and only in the order prescribed in the protocol dances. Additional checks are implemented for each request type and dance step.

C.2 Timestamps, Filestamps and Partial Ordering

When the host starts, it reads the host key and certificate files, which are required for continued operation. It also reads the sign key and leap seconds files, when available. When reading these files the host checks the file formats and filestamps for validity; for instance, all filestamps must be later than the time the UTC timescale was established in 1972 and the certificate filestamp must not be earlier than its associated sign key filestamp. In general, at the time the files are read, the host is not synchronized, so it cannot determine whether the filestamps are bogus other than these simple checks.

In the following the relation $A \rightarrow B$ is Lamport's "happens before" relation, which is true if event *A* happens before event *B*. When timestamps are compared to timestamps, the relation is false if $A \leftrightarrow B$; that is, false if the events are simultaneous. For timestamps compared to filestamps and filestamps compared to filestamps, the relation is true if $A \leftrightarrow B$. Note that the current time plays no part in these assertions except in (6) below; however, the NTP protocol itself insures a correct partial ordering for all current time values.

The following assertions apply to all relevant responses:

1. The client saves the most recent timestamp $T0$ and filestamp $F0$ for the respective signature type. For every received message carrying timestamp $T1$ and filestamp $F1$, the message is discarded unless $T0 \rightarrow T1$ and $F0 \rightarrow F1$. The requirement that $T0 \rightarrow T1$ is the primary defense against replays of old messages.

2. For timestamp T and filestamp F, $T \rightarrow T$; that is, the timestamp must not be earlier than the filestamp. This could be due to a file generation error or a significant error in the system clock time.

3. For sign key filestamp S, certificate filestamp C, cookie timestamp D and autokey timestamp A, $S \rightarrow C \rightarrow D \rightarrow A$; that is, the autokey must be generated after the cookie, the cookie after the certificate and the certificate after the sign key.

4. For sign key filestamp S and certificate filestamp C specifying begin time B and end time E, $S \rightarrow C \rightarrow B \rightarrow E$; that is, the valid period must not be retroactive.

5. A certificate for subject S signed by issuer I and with filestamp $C1$ obsoletes, but does not necessarily invalidate, another certificate with the same subject and issuer but with filestamp $C0$, where $C0 \rightarrow C1$.

6. A certificate with begin time B and end time E is invalid and can not be used to sign certificates if $t \rightarrow B$ or $E \rightarrow t$, where t is the current time. Note that the public key previously extracted from the certificate continues to be valid for an indefinite time. This raises the interesting possibilities where a truechimer server with expired certificate or a falseticker with valid certificate are not detected until the client has synchronized to a clique of proventic truechimers.

D. Security Analysis

This section discusses the most obvious security vulnerabilities in the various Autokey dances. First, some observations on the particular engineering parameters of the Autokey protocol are in order. The number of bits in some cryptographic values are considerably smaller than would ordinarily be expected for strong cryptography. One of the reasons for this is the need for compatibility with previous NTP versions; another is the need for small and constant latencies and minimal processing requirements. Therefore, what the scheme gives up on the strength of these values must be regained by agility in the rate of change of the cryptographic basis values. Thus, autokeys are used only once and seed values are regenerated frequently. However, in most cases even a successful cryptanalysis of these values compromises only a particular association and does not represent a danger to the general population.

Throughout the following discussion the cryptographic algorithms and private values themselves are assumed secure; that is, a brute force crypt analytic attack will not reveal the host private key, sign private key, cookie value, identity parameters, server seed or autokey seed. In addition, an intruder will not be able to predict random generator values or predict the next autokey. On the other hand, the intruder can remember the totality of all past values for all packets ever sent.

D.1 Protocol Vulnerability

While the protocol has not been subjected to a formal analysis, a few preliminary assertions can be made. The protocol cannot loop forever in any state, since the watchdog counter and general reset insure that the association variables will eventually be purged and the protocol restarted from the beginning. However, if something is seriously wrong, the timeout/restart cycle could continue indefinitely until whatever is wrong is fixed. This is not a clogging hazard, as the timeout period is very long compared to expected network delays.

The LBK and DUP bits described in the main body and Appendix C are effective whether or not cryptographic means are in use. The DUP bit deflects duplicate packets in any mode, while the LBK bit deflects bogus packets in all except broadcast mode. All packets must have the correct MAC, as verified with correct key ID and cookie. In all modes the next key ID cannot be predicted by a wiretapper, so are of no use for cryptanalysis.

As long as the client has validated the server certificate trail, a wiretapper cannot produce a convincing signature and cannot produce cryptographic values acceptable to the client. As long as the identity values are not compromised, a middleman cannot masquerade as a legitimate group member and produce convincing certificates or signatures. In server and symmetric modes after the preliminary exchanges have concluded, extension fields are no longer used, a client validates the packet using the autokey sequence. A wiretapper cannot predict the next Key IDs, so cannot produce a valid MAC. A middleman cannot successfully modify and replay a message, since he does not know the cookie and without the cookie cannot produce a valid MAC.

In broadcast mode a wiretapper cannot produce a key list with signed autokey values that a client will accept. The most it can do is replay an old packet causing clients to repeat the autokey hash operations until exceeding the maximum key number. However, a middleman could intercept an otherwise valid broadcast packet and produce a bogus packet with acceptable MAC, since in this case it knows the key ID before the clients do. Of course, the middleman key list would eventually be used up and clients would discover the ruse when verifying the signature of the autokey values. There does not seem to be a suitable defense against this attack.

During the exchanges where extension fields are in use, the cookie is a public value rather than a shared secret and an intruder can easily construct a packet with a valid MAC, but not a valid signature. In the certificate and identity exchanges an intruder can generate fake request messages which may evade server detection; however, the LBK and DUP bits minimize the client exposure to the resulting rogue responses. A wiretapper might be able to intercept a request, manufacture a fake response and loft it swiftly to the client before the real server response. A middleman could do this without even being swift. However, once the identity exchange has completed and the PRV bit lit, these attacks are readily deflected.

A client instantiates cryptographic variables only if the server is synchronized to a proventic source. A server does not sign values or generate cryptographic data files unless synchronized to a proventic source. This raises an interesting issue: how does a client generate proventic cryptographic files before it has ever been synchronized to a proventic source? [Who shaves the barber if the barber shaves everybody in town who does not shave himself?] In principle, this paradox is resolved by assuming the primary (stratum 1) servers are proventicated by external phenomenological means.

D.2 Clogging Vulnerability

There are two clogging vulnerabilities exposed in the protocol design: a encryption attack where the intruder hopes to clog the victim server with needless cookie or signature encryptions or identity calculations, and a decryption attack where the intruder attempts to clog the victim client with needless cookie or verification decryptions. Autokey uses public key cryptography and the algorithms that perform these functions consume significant processor resources.

In order to reduce exposure to decryption attacks the LBK and DUP bits deflect bogus and replayed packets before invoking any cryptographic operations. In order to reduce exposure to encryption attacks, signatures are computed only when the data have changed. For instance, the autokey values are signed only when the key list is regenerated, which happens about once an hour, while the public values are signed only when one of them changes or the server seed is refreshed, which happens about once per day.

In some Autokey dances the protocol precludes server state variables on behalf of an individual client, so a request message must be processed and the response message sent without delay. The identity, cookie and sign exchanges are particularly vulnerable to a clogging attack, since these exchanges can involve expensive cryptographic algorithms as well as digital signatures. A determined intruder could replay identity, cookie or sign requests at high rate, which may very well be a useful munition for an encryption attack. Ordinarily, these requests are seldom used, except when the protocol is restarted or the server seed or public values are refreshed.

Once synchronized to a proventic source, a legitimate extension field with timestamp the same as or earlier than the most recent received of that type is immediately discarded. This foils a middleman cut-and-paste attack using an earlier AUTO response, for example. A legitimate extension field with timestamp in the future is unlikely, as that would require predicting the autokey sequence. In either case the extension field is discarded before expensive signature computations. This defense is most useful in symmetric mode, but a useful redundancy in other modes.

The client is vulnerable to a certificate clogging attack until declared proventic, after which CERT responses are discarded. Before that, a determined intruder could flood the client with bogus certificate responses and force spurious signature verifications, which of course would fail. The intruder could flood the server with bogus certificate requests and cause similar mischief. Once declared proventic, further certificate responses are discard, so these attacks would fail. The intruder could flood the server with replayed sign requests and cause the server to verify the request and sign the response, although the client would drop the response due invalid timestamp.

An interesting adventure is when an intruder replays a recent packet with an intentional bit error. A stateless server will return a crypto-NAK message which the client will notice and discard, since the LBK bit is lit. However, a legitimate crypto-NAK is sent if the server has just refreshed the server seed. In this case the LBK bit is dim and the client performs a general reset and restarts the protocol as expected. Another adventure is to replay broadcast mode packets at high rate. These will be rejected by the clients by the timestamp check and before consuming signature cycles.

n broadcast and symmetric modes the client must include the association ID in the AUTO request. Since association ID values for different invocations of the NTP daemon are randomized over the 16-bit space, it is unlikely that a bogus request would match a valid association with different IP addresses, for example, and cause confusion.

E. Identity Schemes

The Internet infrastructure model described in [1] is based on certificate trails where a subject proves identity to a certificate authority (CA) who then signs the subject certificate using the CA issuer key. The CA in turn proves identity to the next CA and obtains its own signed certificate. The trail continues to a CA with a self-signed trusted root certificate independently validated by other means. If it is possible to prove identity at each step, each certificate along the trail can be considered trusted relative to the identity scheme and trusted root certificate.

The important issue with respect to NTP and ad hoc sensor networks is the cryptographic strength of the identity scheme, since if a middleman could compromise it, the trail would have a security breach. In electric mail and commerce the identity scheme can be based on handwritten signatures, photographs, fingerprints and other things very hard to counterfeit. As applied to NTP subnets and identity proofs, the scheme must allow a client to securely verify that a server knows the same secret that it does, presuming the secret was previously instantiated by secure means, but without revealing the secret to members outside the group.

While the identity scheme described in RFC-2875 [14] is based on a ubiquitous Diffie-Hellman infrastructure, it is expensive to generate and use when compared to others described here. There are five schemes now implemented in Autokey to prove identity: (1) private certificates (PC), (2) trusted certificates (TC), (3) a modified Schnorr algorithm (IFF aka Identify Friendly or Foe), (4) a modified Guillou-Quisquater algorithm (GQ), and (5) a modified Mu-Varadharajan algorithm (MV). The available schemes are selected during the key generation phase, with the particular scheme selected during the parameter exchange. Following is a summary description of each of these schemes.

The IFF, GQ and MV schemes involve a cryptographically strong challenge-response exchange where an intruder cannot learn the group key, even after repeated observations of multiple exchanges. These schemes begin when the client sends a nonce to the server, which then rolls its own nonce, performs a mathematical operation and sends the results along with a message digest to the client. The client performs a second mathematical operation to produce a digest that must match the one included in the message. To the extent that a server can prove identity to a client without either knowing the group key, these schemes are properly described as zero-knowledge proofs.

E.1 Certificates

Certificate extension fields are used to convey information used by the identity schemes, such as whether the certificate is private, trusted or contains a public identity key. While the semantics of these fields generally conforms with conventional usage, there are subtle variations. The fields used by Autokey Version 2 include:

E.1.1 Basic Constraints

This field defines the basic functions of the certificate. It contains the string “critical,CA:TRUE”, which means the field must be interpreted and the associated private key can be used to sign other certificates. While included for compatibility, Autokey makes no use of this field.

E.1.2 Key Usage

This field defines the intended use of the public key contained in the certificate. It contains the string “digitalSignature,keyCertSign”, which means the contained public key can be used to verify signatures on data and other certificates. While included for compatibility, Autokey makes no use of this field.

E.1.3 Extended Key Usage

This field further refines the intended use of the public key contained in the certificate and is present only in self-signed certificates. It contains the string “Private” if the certificate is designated private or the string “trustRoot” if it is designated trusted. A private certificate is always trusted.

E.1.4 Subject Key Identifier:

This field contains the public identity key used in the GQ identity scheme. It is present only if the GQ scheme is configured.

E.2 Private Certificate (PC) Scheme

The PC scheme shown below involves the use of a private certificate as group key. A certificate is designated private by a X509 Version 3 extension field when generated by utility routines in the NTP software distribution. The certificate is distributed to all other group members by secure means and is never revealed inside or outside the group. A client is marked trusted in the Parameter Exchange and authentic when the first signature is verified. This scheme is cryptographically strong as long as the private certificate is protected; however, it can be very awkward to refresh the keys or certificate, since new values must be securely distributed to a possibly large population and activated simultaneously.

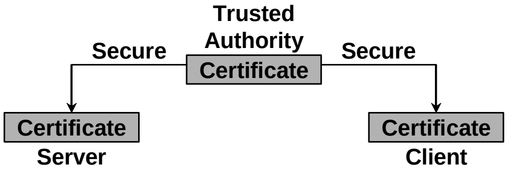

Figure 8. Private Certificate (PC)Identity Scheme

The PC scheme uses a private certificate as group key. A certificate is designated private for the purpose of the this scheme if the CIS Private bit is lit. The certificate is distributed to all other group members by secret means and never revealed outside the group. There is no identity exchange, since the certificate itself is the group key. Therefore, when the parameter exchange completes the VAL, IFF and SGN bits are lit in the server status word. When the following cookie exchange is complete, the PRV bit is lit and operation continues as described in the main body of this document.

E.3 Trusted Certificate (TC) Scheme

All other schemes involve a conventional certificate trail as shown below. As described in RFC-2510, each certificate is signed by an issuer one step closer to the trusted host, which has a self-signed trusted certificate, A certificate is designated trusted by a X509 Version 3 extension field when generated by utility routines in the NTP software distribution. A host obtains the certificates of all other hosts along the trail leading to a trusted host by the Autokey protocol, then requests the immediately ascendant host to sign its certificate. Subsequently, these certificates are provided to descendent hosts by the Autokey protocol. In this scheme keys and certificates can be refreshed at any time, but a masquerade vulnerability remains unless a request to sign a client certificate is validated by some means such as reverse-DNS. If no specific identity scheme is specified in the Identification Exchange, this is the default TC scheme.

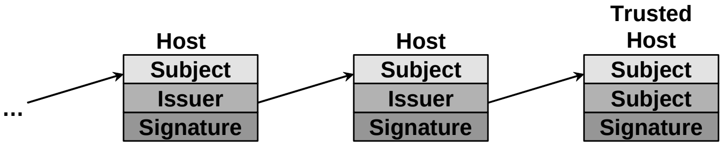

Figure 9. Trusted Certificate (TC) Identity Scheme

The TC identification exchange follows the parameter exchange in which the VAL bit is lit. It involves a conventional certificate trail and a sequence of certificates, each signed by an issuer one stratum level lower than the client, and terminating at a trusted certificate, as described in [1]. A certificate is designated trusted for the purpose of the TC scheme if the CIS Trust bit is lit and the certificate is self-signed. Such would normally be the case when the trail ends at a primary (stratum 1) server, but the trail can end at a secondary server if the security model permits this.

When a certificate is obtained from a server, or when a certificate is signed by a server, A CIS for the new certificate is pushed on the certificate list, but only if the certificate filestamp is greater than any with the same subject name and issuer name already on the list. The list is then scanned looking for signature opportunities. If a CIS issuer name matches the subject name of another CIS and the issuer certificate is verified using the public key of the subject certificate, the Sign bit is lit in the issuer CIS. Furthermore, if the Trust bit is lit in the subject CIS, the Trust bit is lit in the issuer CIS.

The client continues to follow the certificate trail to a self-signed certificate, lighting the Sign and Trust bits as it proceeds. If it finds a self-signed certificate with Trust bit lit, the client lights the IFF and PRV bits and the exchange completes. It can, however, happen that the client finds a self-signed certificate with Trust bit dark. This can happen when a server is just coming up, has synchronized to a proventic source, but has not yet completed the sign exchange. This is considered a temporary condition, so the client simply retries at poll opportunities until the server certificate is signed.

E.4 Schnorr (IFF) Scheme

The Schnorr (IFF) identity scheme is useful when certificates are generated by means other than the NTP software library, such as a trusted public authority. In this case a X.509v3 extension field might not be available to convey the identity public key. The scheme involves a set of parameters which persist for the life of the scheme. New generations of these parameters must be securely transmitted to all members of the group before use. The scheme is self contained and independent of new generations of host keys, sign keys and certificates.

Certificates can be generated by the OpenSSL library or an external public certificate authority, but conveying an arbitrary public value in a certificate extension field might not be possible. The TA generates IFF parameters and keys and distributes them by secure means to all servers, then removes the group key and redistributes these data to dependent clients. Without the group key a client cannot masquerade as a legitimate server.

The IFF parameters are generated by OpenSSL routines normally used to generate DSA keys. By happy coincidence, the mathematical principles on which IFF is based are similar to DSA, but only the moduli *p*, *q* and generator *g* are used in identity calculations. The parameters hide in a RSA cuckoo structure and use the same members. The values are used by an identity scheme based on DSA cryptography and described in [15] and [16] p. 285. The *p* is a 512-bit prime, *g* a generator of the multiplicative group Z_p* and *q* a 160-bit prime that divides $p - 1$ and is a *q*th root of 1 mod *p*; that is, $g^q = 1$ mod *p*. The TA rolls a private random group key *b* ($0 < b < q$), then computes public client key $v = g^{q-b} \bmod p$. The TA distributes (*p*, *q*, *g*, *b*) to all servers using secure means and (*p*, *q*, *g*, *v*) to all clients not necessarily using secure means.

The IFF identity scheme is shown below. The TA generates a DSA parameter structure for use as IFF parameters. The IFF parameters are identical to the DSA parameters, so the OpenSSL library DSA parameter generation routine can be used directly. The DSA parameter structure shown in Table is written to a file as a DSA private key encoded in PEM. Unused structure members are set to one.

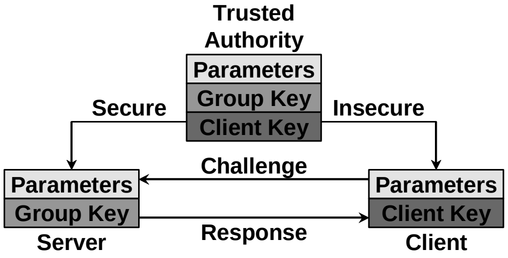

Figure 10. Schnorr (IFF) Identity Scheme

IFF	DSA	Item	Include
p	p	modulus	all
q	q	modulus	all
g	g	generator	all
b	priv_key	group key	server
v	pub_key	client key	client

Table 1. IFF Identity Scheme Parameters

Alice challenges Bob to confirm identity using the following protocol exchange.

1. Alice rolls random *r* ($0 < r < q$) and sends to Bob.

2. Bob rolls random k ($0 < k < q$), computes $y = k + br \bmod q$ and $x = g^k \bmod p$, then sends (y, hash(x)) to Alice.

3. Alice computes $z = g^y v^r \bmod p$ and verifies hash(z) equals hash(x).

If the hashes match, Alice knows that Bob has the group key b. Besides making the response shorter, the hash makes it effectively impossible for an intruder to solve for b by observing a number of these messages. The signed response binds this knowledge to Bob's private key and the public key previously received in his certificate. On success the IFF and PRV bits are lit in the server status word.

E.5 Guillard-Quisquater (GQ) Scheme

The Guillou-Quisquater (GQ) identity scheme is useful when certificates are generated using the NTP software library. These routines convey the GQ public key in a X.509v3 extension field. The scheme involves a set of parameters which persist for the life of the scheme and a private/public identity key, which is refreshed each time a new certificate is generated. The utility inserts the client key in an X.509 extension field when the certificate is generated. The client key is used when computing the response to a challenge. The TA generates the GQ parameters and keys and distributes them by secure means to all group members. The scheme is self contained and independent of new generations of host keys and sign keys and certificates.

The GQ parameters are generated by OpenSSL routines normally used to generate RSA keys. By happy coincidence, the mathematical principles on which GQ is based are similar to RSA, but only the modulus n is used in identity calculations. The parameters hide in a RSA cuckoo structure and use the same members. The values are used in an identity scheme based on RSA cryptography and described in [3] and [16] p. 300 (with errors). The 512-bit public modulus $n = pq$, where p and q are secret large primes. The TA rolls random group key b ($0 < b < n$) and distributes (n, b) to all group members using secure means. The private server key and public client key are constructed later.

The GQ identity scheme is shown below. When generating new certificates, the server rolls new random private server key u ($0 < u < n$) and public client key its inverse obscured by the group key $v = (u^{-1})^b \bmod n$. These values replace the private and public keys normally generated by the RSA scheme. In addition, the public client key is conveyed in a X.509 certificate extension. The updated GQ structure shown in Table is written as a RSA private key encoded in PEM. Unused structure members are set to one.

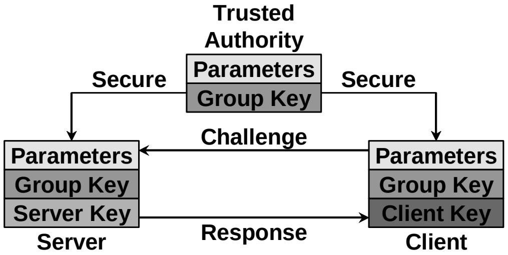

Figure 11. Guillard-Quisquater (GQ) Identity Scheme

GQ	RSA	Item	Include
n	n	modulus	all
b	e	group key	server
u	p	server key	server
v	q	client key	client

Table 2. GQ Identity Scheme Parameters

Alice challenges Bob to confirm identity using the following exchange.

1. Alice rolls random r (0 < r < n) and sends to Bob.

2. Bob rolls random k ($0 < k < n$) and computes $y = ku^r \bmod n$ and $x = k^b \bmod n$, then sends (y, hash(x)) to Alice.

3. Alice computes $z = v^r y^b \bmod n$ and verifies hash(z) equals hash(x).

If the hashes match, Alice knows that Bob has the group key b. Besides making the response shorter, the hash makes it effectively impossible for an intruder to solve for b by observing a number of these messages. The signed response binds this knowledge to Bob's private key and the public key previously received in his certificate. Further evidence is the certificate containing the public identity key, since this is also signed with Bob's private key. On success the IFF and PRV bits are lit in the server status word.

E.6 Mu-Varadharajan (MV) Identity Scheme

The Mu-Varadharajan (MV) scheme was originally intended to encrypt broadcast transmissions to receivers which do not transmit. There is one encryption key for the broadcaster and a separate decryption key for each receiver. It operates something like a pay-per-view satellite broadcasting system where the session key is encrypted by the broadcaster and the decryption keys are held in a tamper proof set-top box. We don't use it this way, but read on.

The MV scheme is perhaps the most interesting and flexible of the three challenge/response schemes. It can be used when a small number of servers provide synchronization to a large client population where there might be considerable risk of compromise between and among the servers and clients. The TA generates an intricate cryptosystem involving public and private encryption keys, together with a number of activation keys and associated private client decryption keys. The activation keys are used by the TA to activate and revoke individual client decryption keys without changing the decryption keys themselves.

The TA provides the server with a private encryption key and public decryption key. The server adjusts the keys by a nonce for each plaintext encryption, so they appear different on each use. The encrypted ciphertext and adjusted public decryption key are provided in the client message. The client computes the decryption key from its private decryption key and the public decryption key in the message.

In the MV scheme the activation keys are known only to the TA. The TA decides which keys to activate and provides to the servers a private encryption key E and public decryption keys $\bar{g}$ and

$\hat{g}$ which depend on the activated keys. The servers have no additional information and, in particular, cannot masquerade as a TA. In addition, the TA provides to each client j individual private decryption keys $\bar{x}_j$ and $\hat{x}_j$, which do not need to be changed if the TA activates or deactivates this key. The clients have no further information and, in particular, cannot masquerade as a server or TA.

The MV values hide in a DSA cuckoo structure which uses the same parameters, but generated in a different way. The values are used in an encryption scheme similar to El Gamal cryptography and a polynomial formed from the expansion of product terms $\prod_{0 < j \le n} (x - x_j)$, as described in [12]. The paper has significant errors and serious omissions.

The MV identity scheme is shown below. The TA writes the server parameters, private encryption key and public decryption keys for all servers as a DSA private key encoded in PEM, as shown in the table below.

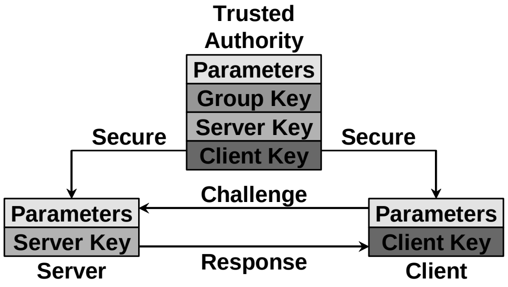

Figure 12. Mu-Varadharajan (MV) Identity Scheme

MV	DSA	Item	Include
p	p	modulus	all
q	q	modulus	server
E	g	private encrypt	server
$\bar{g}$	priv_key	public decrypt	server
$\hat{g}$	pub_key	public decrypt	server

Table 3. MV Scheme Server Parameters

The TA writes the client parameters and private decryption keys for each client as a DSA private key encoded in PEM. It is used only by the designated recipient(s) who pay a suitably outrageous fee for its use. Unused structure members are set to one, as shown in Table.

MV	DSA	Item	Include
p	p	modulus	all
$\bar{x}_j$	priv_key	private decrypt	client
$\hat{x}_j$	pub_key	private decrypt	client

Table 4. MV Scheme Client Parameters

The devil is in the details. Let q be the product of n distinct primes s'_j (j = 1...n), where each s'_j, also called an activation key, has m significant bits. Let prime $p = 2q + 1$, so that q and each s'_j divide $p - 1$ and p has $M = nm + 1$ significant bits. Let g be a generator of the multiplicative group Z_p*; that is, gcd(g, $p - 1$) = 1 and $g^q = 1 \bmod p$. We do modular arithmetic over Z_q and then project into Z_p* as powers of g. Sometimes we have to compute an inverse b^{-1} of random b in Z_q, but for that purpose we require gcd(b, q) = 1. We expect M to be in the 500-bit range and n relatively small, like 30. The TA uses a nasty probabilistic algorithm to generate the cryptosystem.

1. Generate the m-bit primes s'_j $(0 < j \le n)$, which may have to be replaced later. As a practical matter, it is tough to find more than 30 distinct primes for $M \approx 512$ or 60 primes for $M \approx 1024$. The latter can take several hundred iterations and several minutes on a Sun Blade 1000.

2. Compute modulus $q = \prod_{0 < j \le n} s'_j$, then modulus $p = 2q + 1$. If p is composite, the TA replaces one of the primes with a new distinct prime and tries again. Note that q will hardly be a secret since p is revealed to servers and clients. However, factoring q to find the primes should be adequately hard, as this is the same problem considered hard in RSA. Question: is it as hard to find n small prime factors totalling M bits as it is to find two large prime factors totalling M bits? Remember, the bad guy doesn't know n.

3. Associate with each s'_j an element s_j such that $s_j s'_j = s'_j \bmod q$. One way to find an s_j is the quotient $s_j = \frac{q + s'_j}{s'_j}$. The student should prove the remainder is always zero.

4. Compute the generator g of Z_p using a random roll such that $\gcd(g, p-1) = 1$ and $g^q = 1 \bmod p$. If not, roll again.

Once the cryptosystem parameters have been determined, the TA sets up a specific instance of the scheme as follows.

1. Roll n random roots x_j $(0 < x_j < q)$ for a polynomial of order n. While it may not be strictly necessary, Make sure each root has no factors in common with q.

2. Expand the n product terms $\prod_{0 < j \le n} (x - x_j)$ to form n + 1 coefficients a_i mod q $(0 \le i \le n)$ in powers of x using a fast method contributed by C. Boncelet.

3. Generate $g_i = g^{a_i} \bmod p$ for all *i* and the generator *g*. Verify $\prod_{0 \le i \le n,\, 0 < j \le n} g_i^{a_i x_j^{\,i}} = 1 \bmod p$ for all *i, j*. Note the $a_i x_j^{\,i}$ exponent is computed mod *q*, but the g_i is computed mod *p*. Also note the expression given in the paper cited is incorrect.

4. Make master encryption key $A = \prod_{0 < i \le n,\, 0 < j < n} g_i^{x_j} \bmod p$. Keep it around for awhile, since it is expensive to compute.

5. Roll private random group key b (0 < *b* < *q*), where gcd(*b*, *q*) = 1 to guarantee the inverse exists, then compute $b^{-1} \bmod q$. If *b* is changed, all keys must be recomputed.

6. Make private client keys $\bar{x}_j = b^{-1} \sum_{0 < i \le n,\, i \ne j} x_i^{\,n} \bmod q$ and $\hat{x}_j = s_j x_j^{\,n} \bmod q$ for all *j*. Note that the keys for the *j*th client involve only s_j, but not s'_j or *s*. The TA sends (*p*, $\bar{x}_j$, $\hat{x}_j$) to the *j*th client(s) using secure means.

7. The activation key is initially *q* by construction. The TA revokes client *j* by dividing *q* by s'_j. The quotient becomes the activation key *s*. Note we always have to revoke one key; otherwise, the plaintext and cryptotext would be identical. The TA computes $E = A^s$, $\bar{g} = \bar{x}^s \bmod p$, $\hat{g} = \hat{x}^{sb} \bmod p$ and sends (*p*, *E*, $\bar{g}$, $\hat{g}$) to the servers using secure means.

Alice challenges Bob to confirm identity using the following exchange.

1. Alice rolls random *r* (0 < *r* < *q*) and sends to Bob.

2. Bob rolls random *k* (0 < *k* < *q*) and computes the session encryption key $E' = E^k \bmod p$ and public decryption key $\bar{g}' = \bar{g}^k \bmod p$ and $\hat{g}' = \hat{g}^k \bmod p$. He encrypts $x = E'r$ and sends (hash(*x*), $\bar{g}'$, $\hat{g}'$) to Alice.

3. Alice computes the session decryption key $E'^{-1} = \bar{g}'^{\hat{x}_j} \hat{g}'^{\bar{x}_j} \bmod p$, recovers the encryption key $E' = (E'^{-1})^{-1} \bmod p$, encrypts $z = E'r \bmod p$, then verifies that hash(*z*) = hash(*x*).

E.7 Interoperability Issues

A specific combination of authentication scheme (none, symmetric key, Autokey), digest/signature scheme and identity scheme (PC, TC, IFF, GQ, MV) is called a cryptotype, although not all combinations are possible. There may be management configurations where the servers and clients may not all support the same cryptotypes. A secure NTPv4 subnet can be configured in several ways while keeping in mind the principles explained in this section. Note however that some cryptotype combinations may successfully interoperate with each other, but may not represent good security practice.

The cryptotype of an association is determined at the time of mobilization, either at configuration time or some time later when a packet of appropriate cryptotype arrives. When a client, broadcast or symmetric active association is mobilized at configuration time, it can be designated non-authentic, authenticated with symmetric key or authenticated with some Autokey scheme, and subsequently it will send packets with that cryptotype. When a responding server, broadcast client or symmetric passive association is mobilized, it is designated with the same cryptotype as the received packet.

When multiple identity schemes are supported, the parameter exchange determines which one is used. The request message contains bits corresponding to the schemes it supports, while the response message contains bits corresponding to the schemes it supports. The client matches the server bits with its own and selects a compatible identity scheme. The server is driven entirely by the client selection and remains stateless. When multiple selections are possible, the order from most secure to least is GC, IFF, TC. Note that PC does not interoperate with any of the others, since they require the host certificate which a PC server will not reveal.

Following the principle that time is a public value, a server responds to any client packet that matches its cryptotype capabilities. Thus, a server receiving a non-authenticated packet will respond with a non-authenticated packet, while the same server receiving a packet of a cryptotype it supports will respond with packets of that cryptotype. However, new broadcast or manycast client associations or symmetric passive associations will not be mobilized unless the server supports a cryptotype compatible with the first packet received. By default, non-authenticated associations will not be mobilized unless overridden in a decidedly dangerous way.

Some examples may help to reduce confusion. Client Alice has no specific cryptotype selected. Server Bob supports both symmetric key and Autokey cryptography. Alice’s non-authenticated packets arrive at Bob, who replies with non-authenticated packets. Cathy has a copy of Bob’s symmetric key file and has selected key ID 4 in packets to Bob. If Bob verifies the packet with key ID 4, he sends Cathy a reply with that key. If authentication fails, Bob sends Cathy a thing called a crypto-NAK, which tells her something broke. She can see the evidence using the utility programs of the NTP software library.

Symmetric peers Bob and Denise have rolled their own host keys, certificates and identity parameters and lit the host status bits for the identity schemes they can support. Upon completion of the parameter exchange, both parties know the digest/signature scheme and available identity schemes of the other party. They do not have to use the same schemes, but each party must use the digest/signature scheme and one of the identity schemes supported by the other party.

It should be clear from the above that Bob can support all the girls at the same time, as long as he has compatible authentication and identification credentials. Now, Bob can act just like the girls in his own choice of servers; he can run multiple configured associations with multiple different servers (or the same server, although that might not be useful). But, wise security policy might preclude some cryptotype combinations; for instance, running an identity scheme with one server and no authentication with another might not be wise.

F. File Examples

This appendix shows the file formats used by the OpenSSL library and the reference implementation. These are not included in the specification and are given here only as examples. In each case the actual file contents are shown followed by a dump produced by the OpenSSL asn1 program.

F.1 RSA-MD5cert File and ASN.1 Encoding

```
# ntpkey_RSA-MD5cert_whimsy.udel.edu.3236983143
# Tue Jul 30 01:59:03 2002
-----BEGIN CERTIFICATE-----
MIIBkTCCATugAwIBAgIEwPBxZzANBgkqhkiG9w0BAQQFADAaMRgwFgYDVQQDEw93
aGltc3kudWRlbC5lZHUwHhcNMDIwNzMwMDE1OTA3WhcNMDMwNzMwMDE1OTA3WjAa
MRgwFgYDVQQDEw93aGltc3kudWRlbC5lZHUwWjANBgkqhkiG9w0BAQEFAANJADBG
AkEA2PpOz6toSQ3BtdGrBt+F6cSSde6zhayOwRj5nAkOvtQ505hdxWhudfKe7ZOY
HRLLqACvVJEfBaSvE5OFWldUqQIBA6NrMGkwDwYDVR0TAQH/BAUwAwEB/zALBgNV
HQ8EBAMCAoQwSQYDVR0OBEIEQEVFGZar3afoZcHDmhbgiOmaBrtWTlLHRwIJswge
LuqB1fbsNEgUqFebBR1Y9qLwYQUm7ylBD+3z9PlhcUOwtnIwDQYJKoZIhvcNAQEE
BQADQQAVZMiNbYV2BjvFH9x+t0PB9//giOV3fQoLK8hXXpyiAF4KLleEqP13pK0H
TceF3e3bxSRTndkIhklEAcbYXm66
-----END CERTIFICATE-----
```

```
  0:d=0  hl=4 l= 401 cons: SEQUENCE
  4:d=1  hl=4 l= 315 cons: SEQUENCE
  8:d=2  hl=2 l=   3 cons: cont [ 0 ]
 10:d=3  hl=2 l=   1 prim: INTEGER:02
 13:d=2  hl=2 l=   4 prim: INTEGER :-3F0F8E99
 19:d=2  hl=2 l=  13 cons: SEQUENCE
 21:d=3  hl=2 l=   9 prim: OBJECT:md5WithRSAEncryption
 32:d=3  hl=2 l=   0 prim: NULL
 34:d=2  hl=2 l=  26 cons: SEQUENCE
 36:d=3  hl=2 l=  24 cons: SET
 38:d=4  hl=2 l=  22 cons: SEQUENCE
 40:d=5  hl=2 l=   3 prim: OBJECT:commonName
 45:d=5  hl=2 l=  15 prim: PRINTABLESTRING :whimsy.udel.edu
 62:d=2  hl=2 l=  30 cons: SEQUENCE
 64:d=3  hl=2 l=  13 prim: UTCTIME:020730015907Z
 79:d=3  hl=2 l=  13 prim: UTCTIME:030730015907Z
 94:d=2  hl=2 l=  26 cons: SEQUENCE
 96:d=3  hl=2 l=  24 cons: SET
 98:d=4  hl=2 l=  22 cons: SEQUENCE
100:d=5  hl=2 l=   3 prim: OBJECT:commonName
105:d=5  hl=2 l=  15 prim: PRINTABLESTRING :whimsy.udel.edu
122:d=2  hl=2 l=  90 cons: SEQUENCE
124:d=3  hl=2 l=  13 cons: SEQUENCE
126:d=4  hl=2 l=   9 prim: OBJECT:rsaEncryption
137:d=4  hl=2 l=   0 prim: NULL
139:d=3  hl=2 l=  73 prim: BIT STRING
214:d=2  hl=2 l= 107 cons: cont [ 3 ]
216:d=3  hl=2 l= 105 cons: SEQUENCE
218:d=4  hl=2 l=  15 cons: SEQUENCE
220:d=5  hl=2 l=   3 prim: OBJECT:X509v3 Basic Constraints
225:d=5  hl=2 l=   1 prim: BOOLEAN:255
228:d=5  hl=2 l=   5 prim: OCTET STRING
```

```
235:d=4  hl=2 l=  11 cons: SEQUENCE
237:d=5  hl=2 l=   3 prim: OBJECT:X509v3 Key Usage
242:d=5  hl=2 l=   4 prim: OCTET STRING
248:d=4  hl=2 l=  73 cons: SEQUENCE
250:d=5  hl=2 l=   3 prim: OBJECT:X509v3 Subject Key Identifier
255:d=5  hl=2 l=  66 prim: OCTET STRING
323:d=1  hl=2 l=  13 cons: SEQUENCE
325:d=2  hl=2 l=   9 prim: OBJECT:md5WithRSAEncryption
336:d=2  hl=2 l=   0 prim: NULL
338:d=1  hl=2 l=  65 prim: BIT STRING
```

F.2 RSAkey File and ASN.1 Encoding

```
# ntpkey_RSAkey_whimsy.udel.edu.3236983143
# Tue Jul 30 01:59:03 2002
-----BEGIN RSA PRIVATE KEY-----
MIIBOgIBAAJBANj6Ts+raEkNwbXRqwbfhenEknXus4WsjsEY+ZwJDr7UOdOYXcVo
bnXynu2TmB0Sy6gAr1SRHwWkrxOThVpXVKkCAQMCQQCQpt81HPAws9Z5NnIElQPx
Lbb5Sc0DyF8rZfu9W18p4Zb5UH3KYqZfAO4K0GTmxuriFphgS9bELSw5L6ow4t6D
AiEA7ACLlKZtCp91CaDohViPhs7KBdRVq7DG9n88z9MM/gMCIQDrXRQMb2dqR/ww
PHJ7aljkhhTE78mxLpn2Po82PfYI4wIhAJ1VsmMZngcU+LEV8FjltQSJ3APi48fL
L07/fd/iCKlXAiEAnOi4CEpE8YVSytL2/PGQmFljLfUxIMm7+X8KJClOsJcCICgU
1w07kRO2ycicL2QRVh8J8vQL68VfH53H+oobKDCd
-----END RSA PRIVATE KEY-----
```

```
  0:d=0  hl=4 l= 314 cons: SEQUENCE
  4:d=1  hl=2 l=   1 prim: INTEGER:00
  7:d=1  hl=2 l=  65 prim: INTEGER:<hex string omitted>
 74:d=1  hl=2 l=   1 prim: INTEGER:03
 77:d=1  hl=2 l=  65 prim: INTEGER:<hex string omitted>
144:d=1  hl=2 l=  33 prim: INTEGER:<hex string omitted>
179:d=1  hl=2 l=  33 prim: INTEGER:<hex string omitted>
214:d=1  hl=2 l=  33 prim: INTEGER:<hex string omitted>
249:d=1  hl=2 l=  33 prim: INTEGER:<hex string omitted>
284:d=1  hl=2 l=  32 prim: INTEGER:<hex string omitted>
```

F.3 IFFpar File and ASN.1 Encoding

```
# ntpkey_IFFpar_whimsy.udel.edu.3236983143
# Tue Jul 30 01:59:03 2002
-----BEGIN DSA PRIVATE KEY-----
MIH4AgEAAkEA7fBvqq9+3DH5BnBScMkruqH4QEB76oec1zjWQ23gyoP2U+L8tHfv
z2LmogOqE1c0McgQynyfQMSDUEmxMyiDwQIVAJ18qdV84wmiCGmWgsHKbpAwepDX
AkA4y42QqZ8aUzQRwkMuYTKbyRRnCG1TJi5eVJcCq65twl5c1bnn24xkbl+FXqck
G6w9NcDtSzuYg1gFLxEuWsYaAkEAjc+nPJR7VY4BjDleVTna07edDfcySl9vy8Pa
B4qArk51LdJlJ49yxEPUxFy/KBIFEHCwRZMc1J7z7dQ/Af26zQIUMXkbVz0D+2Yo
YlG0C/F33Q+N5No=
-----END DSA PRIVATE KEY-----
```

```
  0:d=0  hl=3 l= 248 cons: SEQUENCE
  3:d=1  hl=2 l=   1 prim: INTEGER:00
  6:d=1  hl=2 l=  65 prim: INTEGER:<hex string omitted>
 73:d=1  hl=2 l=  21 prim: INTEGER:<hex string omitted>
 96:d=1  hl=2 l=  64 prim: INTEGER:<hex string omitted>
```

```
162:d=1  hl=2 l=  65 prim: INTEGER:<hex string omitted>
229:d=1  hl=2 l=  20 prim: INTEGER:<hex string omitted>
```

G. ASN.1 Encoding Rules

Certain value fields in request and response messages contain data encoded in ASN.1 distinguished encoding rules (DER). The BNF grammer for each encoding rule is given below along with the OpenSSL routine used for the encoding in the reference implementation. The object identifiers for the encryption algorithms and message digest/signature encryption schemes are specified in [2]. The particular algorithms required for conformance are not specified in this document.

G.1 COOKIE request, IFF response, GQ response, MV response

The value field of the COOKIE request message contains a sequence of two integers (*n*, *e*) encoded by the `i2d_RSAPublicKey()` routine in the OpenSSL distribution. In the request, *n* is the RSA modulus in bits and *e* is the public exponent.

```
RSAPublicKey ::= SEQUENCE {
   n ::= INTEGER,
   e ::= INTEGER
}
```

The IFF and GQ responses contain a sequence of two integers (*r*, *s*) encoded by the `i2d_DSA_SIG()` routine in the OpenSSL distribution. In the responses, *r* is the challenge response and *s* is the hash of the private value.

```
DSAPublicKey ::= SEQUENCE {
   r ::= INTEGER,
   s ::= INTEGER
}
```

The MV response contains a sequence of three integers (*p*, *q*, *g*) encoded by the `i2d_DSAparams()` routine in the OpenSSL library. In the response, *p* is the hash of the encrypted challenge value and (*q*, *g*) is the client portion of the decryption key.

```
DSAparameters ::= SEQUENCE {
   p ::= INTEGER,
   q ::= INTEGER,
   g ::= INTEGER
}
```

G.2 CERT response, SIGN request and response

The value field contains a X509v3 certificate encoded by the `i2d_X509()` routine in the OpenSSL distribution. The encoding follows the rules stated in [4], including the use of X509v3 extension fields.

```
Certificate ::= SEQUENCE {
   tbsCertificate                  TBSCertificate,
   signatureAlgorithm              AlgorithmIdentifier,
   signatureValue                  BIT STRING
}
```

The signatureAlgorithm is the object identifier of the message digest/signature encryption scheme used to sign the certificate. The signatureValue is computed by the certificate issuer using this algorithm and the issuer private key.

```
TBSCertificate ::= SEQUENCE {
   version                          EXPLICIT v3(2),
   serialNumber                     CertificateSerialNumber,
   signature                        AlgorithmIdentifier,
   issuer                           Name,
   validity                         Validity,
   subject                          Name,
   subjectPublicKeyInfo             SubjectPublicKeyInfo,
   extensions                       EXPLICIT Extensions OPTIONAL
}
```

The serialNumber is an integer guaranteed to be unique for the generating host. The reference implementation uses the NTP seconds when the certificate was generated. The signature is the object identifier of the message digest/signature encryption scheme used to sign the certificate. It must be identical to the signatureAlgorithm.

```
CertificateSerialNumber ::= INTEGER
Validity ::= SEQUENCE {
   notBefore                        UTCTime,
   notAfter                         UTCTime
}
```

The notBefore and notAfter define the period of validity as defined in Appendix E.

```
SubjectPublicKeyInfo ::= SEQUENCE {
   algorithm                        AlgorithmIdentifier,
   subjectPublicKey                 BIT STRING
}
```

The AlgorithmIdentifier specifies the encryption algorithm for the subject public key. The subjectPublicKey is the public key of the subject.

```
Extensions ::= SEQUENCE SIZE (1..MAX) OF Extension
Extension ::= SEQUENCE {
   extnID                           OBJECT IDENTIFIER,
   critical                         BOOLEAN DEFAULT FALSE,
   extnValue                        OCTET STRING
}

Name ::= SEQUENCE {
   OBJECT IDENTIFIER                commonName
   PrintableString                  HostName
}
```

For all certificates, the subject HostName is the unique DNS name of the host to which the public key belongs. The reference implementation uses the string returned by the Unix gethostname() routine (trailing NUL removed). For other than self-signed certificates, the issuer HostName is the unique DNS name of the host signing the certificate.