Network Time Protocols
Internet-Draft
Intended status: Experimental
Expires: 24 October 2025

W. Ladd
Akamai Technologies
M. Dansarie
Netnod

22 April 2025

Roughtime
draft-ietf-ntp-roughtime-14

Abstract

This document describes Roughtime — a protocol that aims to achieve two things: secure rough time synchronization even for clients without any idea of what time it is, and giving clients a format by which to report any inconsistencies they observe between time servers. This document specifies the on-wire protocol required for these goals, and discusses aspects of the ecosystem needed for it to work.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-ntp-roughtime/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-ntp/draft-roughtime.

Status of This Memo

Table of Contents

1. Introduction

Time synchronization is essential to Internet security as many security protocols and other applications require synchronization [RFC738]. Unfortunately, widely deployed protocols such as the Network Time Protocol (NTP) [RFC5905] lack essential security features, and even newer protocols like Network Time Security (NTS) [RFC8915] lack mechanisms to observe that the servers behave correctly. Furthermore, clients may lack even a basic idea of the time, creating bootstrapping problems.

The primary design goal of Roughtime is to permit devices to obtain a rough idea of the current time from fairly static configuration and to enable them to report any inconsistencies they observe between servers. The configuration consists of a list of servers and their associated long-term keys, which ideally remain unchanged throughout a server's lifetime. This makes the long-term public keys the roots of trust in Roughtime. With a sufficiently long list of trusted servers and keys, a client will be able to acquire authenticated time with high probability, even after long periods of inactivity. Proofs of malfeasance constructed by chaining together responses from different trusted servers can be used to prove misbehavior by a server, thereby revoking trust in that particular key.

This memo is limited to describing the Roughtime on-wire protocol. Apart from describing the server list and malfeasance report formats, this memo does not describe the ecosystem required for maintaining lists of trusted servers and processing malfeasance reports.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Protocol Overview

Roughtime is a protocol for authenticated rough time synchronization that enables clients to provide cryptographic proof of server malfeasance. It does so by having responses from servers include a signature over a value derived from the client's request, which includes a nonce. This provides cryptographic proof that the timestamp was issued after the server received the client's request. The derived value included in the server's response is the root of a Merkle tree [Merkle] which includes the hash value of the client's request as the value of one of its leaf nodes. This enables the server to amortize the relatively costly signing operation over a number of client requests.

3.1. Single Server Mode

At its most basic level, Roughtime is a one round protocol in which a completely fresh client requests the current time and the server sends a signed response. The response includes a timestamp and a radius used to indicate the server's certainty about the reported time.

The client’s request contains a nonce which the server incorporates into its signed response. The client can verify the server’s signatures and— provided that the nonce has sufficient entropy— this proves that the signed response could only have been generated after the nonce.

3.2. Multi Server Mode

When using multiple servers, a client can detect, cryptographically prove, and report inconsistencies between different servers.

A Roughtime server guarantees that the timestamp included in the response to a query is generated after the reception of the query and prior to the transmission of the associated response. If the time response from a server is not consistent with time responses from other servers, this indicates server error or intentional malfeasance that can be reported and potentially used to impeach the server.

Proofs of malfeasance are constructed by chaining requests to different Roughtime servers. Details on proofs and malfeasance reporting are provided in Section 8. For the reporting to result in impeachment, an additional mechanism is required that provides a review and impeachment process. Defining such a mechanism is beyond the scope of this document. A simple option could be an online forum where a court of human observers judge cases after reviewing input reports.

4. Message Format

Roughtime messages are maps consisting of one or more (tag, value) pairs. They start with a header, which contains the number of pairs, the tags, and value offsets. The header is followed by a message values section which contains the values associated with the tags in the header. Messages MUST be formatted according to Figure 1 as described in the following sections.

Messages MAY be recursive, i.e. the value of a tag can itself be a Roughtime message.

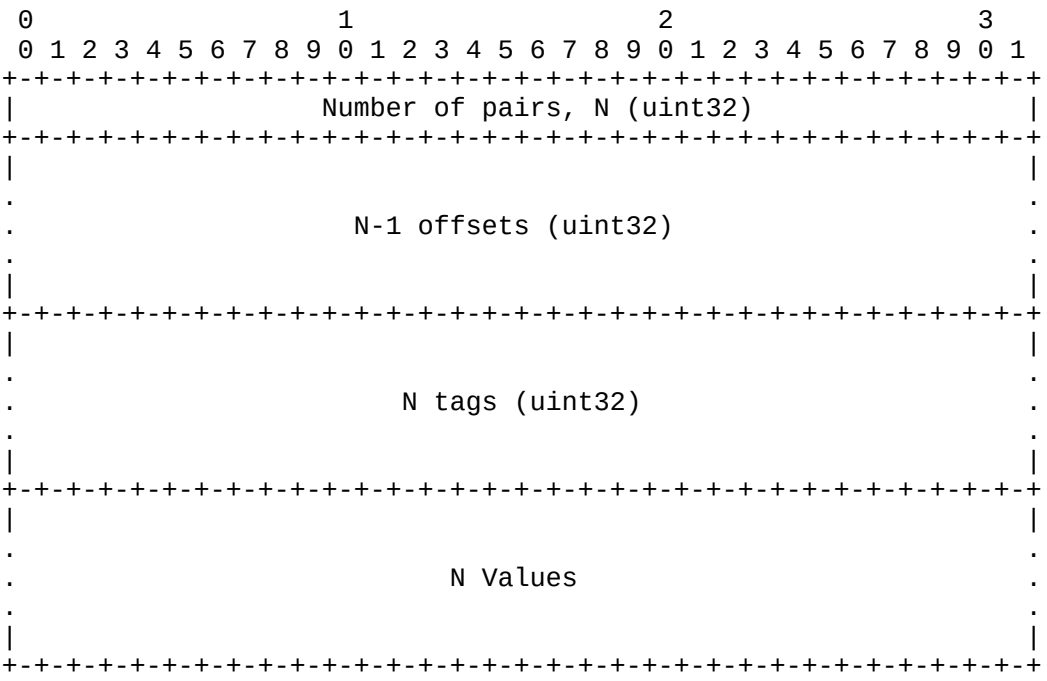

Figure 1: Roughtime Message

4.1. Data types

4.1.1. uint32

A uint32 is a 32 bit unsigned integer. It is serialized with the least significant byte first.

4.1.2. uint64

A uint64 is a 64 bit unsigned integer. It is serialized with the least significant byte first.

4.1.3. Tag

Tags are used to identify values in Roughtime messages. A tag is a uint32 but can also be represented as a sequence of up to four ASCII characters [RFC20] with the first character in the most significant byte. ASCII strings shorter than four characters can be unambiguously converted to tags by padding them with zero bytes. Tags MUST NOT contain any other bytes than capital letters (A-Z) or padding zero bytes. For example, the ASCII string "NONC" would correspond to the tag 0x434e4f4e and "VER" would correspond to

0x00524556.

4.1.4. Timestamp

A timestamp is a representation of UTC time as a uint64 count of seconds since 00:00:00 on 1 January 1970 (the Unix epoch), assuming every day has 86400 seconds. This is a constant offset from the NTP timestamp in seconds. Leap seconds do not have an unambiguous representation in a timestamp, and this has implications for the attainable accuracy and setting of the RADI tag.

4.2. Header

All Roughtime messages start with a header. The first four bytes of the header is the uint32 number of tags N, and hence of (tag, value) pairs.

The following 4*(N-1) bytes are offsets, each a uint32. The last 4*N bytes in the header are tags. Offsets refer to the positions of the values in the message values section. All offsets MUST be multiples of four and placed in increasing order. The first post-header byte is at offset 0. The offset array is considered to have a not explicitly encoded value of 0 as its zeroth entry.

The value associated with the ith tag begins at offset[i] and ends at offset[i+1]-1, with the exception of the last value which ends at the end of the message. Values may have zero length. All lengths and offsets are in bytes.

Tags MUST be listed in the same order as the offsets of their values and be sorted in ascending order by numeric value. A tag MUST NOT appear more than once in a header.

5. Protocol Details

As described in Section 3, clients initiate time synchronization by sending requests containing a nonce to servers who send signed time responses in return. Roughtime packets can be sent between clients and servers either as UDP datagrams or via TCP streams. Servers SHOULD support both the UDP and TCP transport modes.

A Roughtime packet MUST be formatted according to Figure 2 and as described here. The first field is a uint64 with the value 0x4d49544847554f52 ("ROUGHTIM" in ASCII). The second field is a uint32 and contains the length of the third field. The third and last field contains a Roughtime message as specified in Section 4.

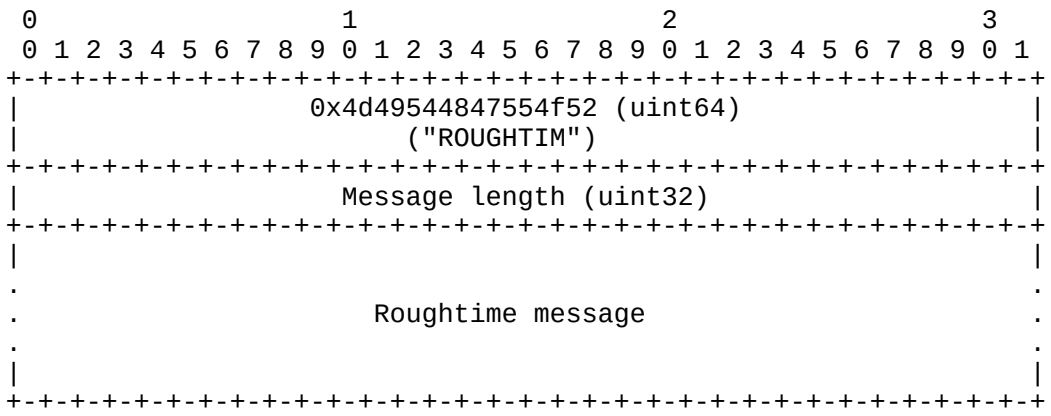

Figure 2: Roughtime packet

Roughtime request and response packets MUST be transmitted in a single datagram when the UDP transport mode is used. Setting the packet's don't fragment bit [RFC791] is OPTIONAL in IPv4 networks.

Multiple requests and responses can be exchanged over an established TCP connection. Clients MAY send multiple requests at once and servers MAY send responses out of order. The connection SHOULD be closed by the client when it has no more requests to send and has received all expected responses. Either side SHOULD close the connection in response to synchronization, format, implementation-defined timeouts, or other errors.

All requests and responses MUST contain the VER tag. It contains a list of one or more uint32 version numbers. The version of Roughtime specified by this memo has version number 1.

NOTE TO RFC EDITOR: remove this paragraph before publication. For testing this draft of the memo, a version number of 0x8000000c is used.

5.1. Requests

A request MUST contain the tags VER, NONC, and TYPE. It SHOULD include the tag SRV. Other tags SHOULD be ignored by the server. Requests not containing the three mandatory tags MUST be ignored by servers. A future version of this protocol may mandate additional tags in the message and assign them semantic meaning.

The size of the request message SHOULD be at least 1024 bytes when the UDP transport mode is used. To attain this size the ZZZZ tag SHOULD be added to the message. Responding to requests shorter than 1024 bytes is OPTIONAL and servers MUST NOT send responses larger than the requests they are replying to, see Section 9.7.

5.1.1. VER

In a request, the VER tag contains a list of uint32 version numbers. The VER tag MUST include at least one Roughtime version supported by the client and MUST NOT contain more than 32 version numbers. The client MUST ensure that the version numbers and tags included in the request are not incompatible with each other or the packet contents.

The version numbers MUST NOT repeat and MUST be sorted in ascending numerical order.

Servers SHOULD ignore any unknown version numbers in the list supplied by the client. If the list contains no version numbers supported by the server, it MAY respond with another version or ignore the request entirely, see Section 5.2.5.

5.1.2. NONC

The value of the NONC tag is a 32-byte nonce. It SHOULD be generated in a manner indistinguishable from random. BCP 106 [RFC4086] contains specific guidelines regarding this.

5.1.3. TYPE

The TYPE tag is used to unambiguously distinguish between request and response messages. In a request, it MUST contain a uint32 with value 0. Requests containing a TYPE tag with any other value MUST be ignored by servers.

5.1.4. SRV

The SRV tag is used by the client to indicate which long-term public key it expects to verify the response with. The value of the SRV tag is H(0xff || public_key) where public_key is the server's long-term, 32-byte Ed25519 public key and H is SHA-512 truncated to the first 32 bytes.

5.1.5. ZZZZ

The ZZZZ tag is used to expand the response to the minimum required length. Its value MUST be all zero bytes.

5.2. Responses

The server begins the request handling process with a set of long-term keys. It resolves which long-term key to use with the following procedure:

1. If the request contains a SRV tag, then the server looks up the long-term key indicated by the SRV value. If no such key exists, then the server MUST ignore the request.

2. If the request contains no SRV tag, but the server has just one long-term key, it SHOULD select that key. Otherwise, if the server has multiple long-term keys, then it MUST ignore the request.

A response MUST contain the tags SIG, NONC, TYPE, PATH, SREP, CERT, and INDX. The structure of a response message is illustrated in Figure 3.

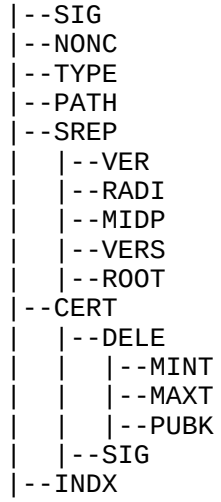

Figure 3: Roughtime response message structure.

5.2.1. SIG

In general, a SIG tag value is a 64-byte Ed25519 signature [RFC8032] over a concatenation of a signature context ASCII string and the entire value of a tag. All context strings MUST include a terminating zero byte.

The SIG tag in the root of a response MUST be a signature over the SREP value using the public key contained in CERT. The context string MUST be "RoughTime v1 response signature".

5.2.2. NONC

The NONC tag MUST contain the nonce of the message being responded to.

5.2.3. TYPE

In a response, the TYPE tag MUST contain a uint32 with value 1. Responses containing a TYPE tag with any other value MUST be ignored by clients.

5.2.4. PATH

The PATH tag value MUST be a multiple of 32 bytes long and represent a path of 32-byte hash values in the Merkle tree used to generate the ROOT value as described in a Section 5.3. In the case where a response is prepared for a single request and the Merkle tree contains only the root node, the size of PATH MUST be zero.

The PATH MUST NOT contain more than 32 hash values. The maximum length of PATH is normally limited by the maximum size of the response message, see Section 5.1 and Section 9.7. Server implementations SHOULD select a maximum Merkle tree height (see Section 5.3) that ensures this.

5.2.5. SREP

The SREP tag contains a signed response. Its value MUST be a Roughtime message with the tags VER, RADI, MIDP, VERS, and ROOT.

The VER tag, when used in a response, MUST contain a single uint32 version number. It SHOULD be one of the version numbers supplied by the client in its request. The server MUST ensure that the version number corresponds with the rest of the packet contents.

The RADI tag value MUST be a uint32 representing the server's estimate of the accuracy of MIDP in seconds. Servers MUST ensure that the true time is within (MIDP-RADI, MIDP+RADI) at the moment of processing. The value of RADI MUST NOT be zero. Since leap seconds can not be unambiguously represented by Roughtime timestamps, servers MUST take this into account when setting the RADI value during leap second events. Servers that do not have any leap second information SHOULD set the value of RADI to at least 3. Failure to do so will impact the observed correctness of Roughtime servers and can lead to malfeasance reports.

The MIDP tag value MUST be the timestamp of the moment of processing.

The VERS tag value MUST contain a list of uint32 version numbers supported by the server, sorted in ascending numerical order. It MUST contain the version number specified in the VER tag. It MUST NOT contain more than 32 version numbers.

The ROOT tag MUST contain a 32-byte value of a Merkle tree root as described in Section 5.3.

5.2.6. CERT

The CERT tag contains a public-key certificate signed with the server's private long-term key. Its value MUST be a Roughtime message with the tags DELE and SIG, where SIG is a signature over the DELE value. The context string used to generate SIG MUST be "RoughTime v1 delegation signature".

The DELE tag contains a delegated public-key certificate used by the server to sign the SREP tag. Its value MUST be a Roughtime message with the tags MINT, MAXT, and PUBK. The purpose of the DELE tag is to enable separation of a long-term public key from keys on devices exposed to the public Internet.

The MINT tag is the minimum timestamp for which the key in PUBK is trusted to sign responses. MIDP MUST be more than or equal to MINT for a response to be considered valid.

The MAXT tag is the maximum timestamp for which the key in PUBK is trusted to sign responses. MIDP MUST be less than or equal to MAXT for a response to be considered valid.

The PUBK tag MUST contain a temporary 32-byte Ed25519 public key which is used to sign the SREP tag.

5.2.7. INDX

The INDX tag value MUST be a uint32 determining the position of NONC in the Merkle tree used to generate the ROOT value as described in Section 5.3.

5.3. The Merkle Tree

A Merkle tree [Merkle] is a binary tree where the value of each non-leaf node is a hash value derived from its two children. The root of the tree is thus dependent on all leaf nodes.

In Roughtime, each leaf node in the Merkle tree represents one request. Leaf nodes are indexed left to right, beginning with zero.

The values of all nodes are calculated from the leaf nodes and up towards the root node using the first 32 bytes of the output of the SHA-512 hash algorithm [RFC6234]. For leaf nodes, the byte 0x00 is prepended to the full value of the client's request packet, including the "ROUGHTIM" header, before applying the hash function. For all other nodes, the byte 0x01 is concatenated with first the left and then the right child node value before applying the hash function.

The value of the Merkle tree's root node is included in the ROOT tag of the response.

The index of a request leaf node is included in the INDX tag of the response.

The values of all sibling nodes in the path between a request leaf node and the root node are stored in the PATH tag so that the client can reconstruct and validate the value in the ROOT tag using its request packet. These values are each 32 bytes and are stored one after the other with no additional padding or structure. The order in which they are stored is described in the next section.

5.3.1. Root Value Validity Check Algorithm

This section describes how to compute the value of the root of the Merkle tree from the values in the tags PATH, INDX, and NONC. The bits of INDX are ordered from least to most significant. H(x) denotes the first 32 bytes of the SHA-512 hash digest of x and || denotes concatenation.

The algorithm maintains a current value h. At initialization, h is set to H(0x00 || request_packet). When no more entries remain in PATH, h is the value of the root of the Merkle tree. All remaining bits of INDX MUST be zero at that time. Otherwise, let node be the next 32 bytes in PATH. If the current bit in INDX is 0 then h = H(0x01 || node || hash), else h = H(0x01 || hash || node).

PATH is thus the siblings from the leaf to the root.

5.4. Validity of Response

A client MUST check the following properties when it receives a response. We assume the long-term server public key is known to the client through other means.

The signature in CERT was made with the long-term key of the server.

The MIDP timestamp lies in the interval specified by the MINT and MAXT timestamps.

The INDX and PATH values prove a hash value derived from the request packet was included in the Merkle tree with value ROOT using the algorithm in Section 5.3.1.

The signature of SREP in SIG validates with the public key in DELE.

A response that passes these checks is said to be valid. Validity of a response does not prove that the timestamp’s value in the response is correct, but merely that the server guarantees that it signed the timestamp and computed its signature during the time interval (MIDP-RADI, MIDP+RADI).

6. Integration into NTP

We assume that there is a bound PHI on the frequency error in the clock on the machine. Let delta be the time difference between the clock on the client and the clock on the server, and let sigma represent the error in the measured value of delta introduced by the measurement process.

Given a measurement taken at a local time t, we know the true time is in (t-delta-sigma, t-delta+sigma). After d seconds have elapsed we know the true time is within (t-delta-sigma-d_PHI, t-delta+sigma+d_PHI). A simple and effective way to mix with NTP or Precision Time Protocol (PTP) discipline of the clock is to trim the observed intervals in NTP to fit entirely within this window or reject measurements that fall too far outside. This assumes time has not been stepped. If the NTP process decides to step the time, it

MUST use Roughtime to ensure the new true time estimate that will be stepped to is consistent with the true time. Should this window become too large, another Roughtime measurement is called for. The definition of "too large" is implementation defined. Implementations MAY use other, more sophisticated means of adjusting the clock respecting Roughtime information. Other applications such as X.509 verification may wish to apply different rules.

If an NTP server uses a Roughtime server as a time source for synchronisation (and not only for filtering its NTP measurements), the root dispersion SHOULD include the server's RADI value and root delay SHOULD include the interval between sending the Roughtime request and receiving the response.

7. Grease

The primary purpose of grease is to prevent protocol ossification, which could prohibit future protocol extensions and development [RFC9170]. In Roughtime, grease is also intended to ensure that clients validate signatures. To grease the Roughtime protocol, servers SHOULD send back a fraction of responses with any of the following: lack of mandatory tags, version numbers not in the request, undefined tags, or invalid signatures together with incorrect times. Clients MUST properly ignore undefined tags and reject invalid responses. Servers MUST NOT send back responses with incorrect times and valid signatures. Either signature MAY be invalid for this application.

8. Roughtime Clients

8.1. Necessary configuration

To carry out a Roughtime measurement, a client SHOULD be equipped with a list of servers, a minimum of three of which are operational and not run by the same parties. Roughtime clients SHOULD regularly update their view of which servers are trustworthy in order to benefit from the detection of misbehavior. Clients SHOULD also have a means of reporting to the provider of such a list, such as an operating system or software vendor, a malfeasence report as described below.

8.2. Measurement Sequence

The client randomly selects at least three servers from the list, and sequentially queries them. The query sequence SHOULD be repeated twice with the servers in the same order, to ensure that all possible inconsistences can be detected.

The first probe uses a nonce that is randomly generated. The second query uses H(resp || rand) where rand is a random 32-byte value and resp is the entire response to the first probe, including the "ROUGHTIM" header. Each subsequent query uses H(resp || rand) for the previous response and a different 32-byte rand value. H(x) and || are defined as in Section 5.3.1.

For each pair of responses (i, j), where i was received before j, the client MUST check that MIDP_i-RADI_i is less than or equal to MIDP_j+RADI_j. If all checks pass, the times are consistent with causal ordering. If at least one check fails, there has been a malfeasance and the client SHOULD store a report for evaluation, alert the user, and make another measurement. If the times reported are consistent with the causal ordering, and the delay between request and response is within an implementation-dependent maximum value, the measurement succeeds.

8.3. Server Lists

To facilitate regular updates of lists of trusted servers, clients SHOULD implement the server list format specified here. Server lists MUST be formatted as JSON [RFC8259] objects and contain the key "servers". Client lists MAY also contain the keys "sources" and "reports".

The value of the "servers" key MUST be a list of server objects, each containing the keys "name", "version", "publicKeyType", "publicKey", and "addresses".

The value of "name" MUST be a string and SHOULD contain a server name suitable for display to a user.

The value of "version" MUST be an integer that indicates the highest Roughtime version number supported by the server.

NOTE TO RFC EDITOR: remove this paragraph before publication. To indicate compatibility with drafts of this memo, a decimal representation of the version number indicated in Section 5 SHOULD be used. For indicating compatibility with pre-IETF specifications of Roughtime, the version number 3000600613 SHOULD be used.

The value of "publicKeyType" MUST be a string indicating the signature scheme used by the server. The value for servers supporting version 1 of Roughtime MUST be "ed25519".

The value of "publicKey" MUST be a base64-encoded [RFC4648] string representing the long-term public key of the server in a format consistent with the value of "publicKeyType".

The value of "addresses" MUST be a list of address objects. An address object MUST contain the keys "protocol" and "address". The value of "protocol" MUST be either "tcp" or "udp", indicating the transport mode to use. The value of "address" MUST be string indicating a host and a port number, separated by a colon character, for example "roughtime.example.com:2002". The host part SHALL be either an IPv4 address, an IPv6 address, or a fully qualified domain name (FQDN). IPv4 addresses MUST be in dotted decimal notation. IPv6 addresses MUST conform to the "Text Representation of Addresses" [RFC4291] and MUST NOT include zone identifiers [RFC6874]. The port part SHALL be a decimal integer representing a valid port number, i.e. in the range 0-65535.

The value of "sources", if present, MUST be a list of strings indicating where updated versions of the list may be aquired. Each string MUST be a URL [RFC1738] pointing to a list in the format specified here. The URI scheme MUST be HTTPS [RFC9110].

The value of "reports", if present, MUST be a string indicating a URL [RFC1738] where malfeasance reports can be sent by clients using the HTTP POST method [RFC9110]. The URI scheme MUST be HTTPS [RFC9110].

8.4. Malfeasance Reporting

A malfeasance report is cryptographic proof that a sequence of responses arrived in that order. It can be used to demonstrate that at least one server sent the wrong time.

A malfeasance report MUST be formatted as a JSON [RFC8259] object and contain the key "responses". Its value MUST be an ordered list of response objects. Each response object MUST contain the keys "rand", "request", "response", and "publicKey". The values of all four keys MUST be represented as base64-encoded [RFC4648] strings.

The "rand" key MAY be omitted from the first response object in the list. In all other cases, its value MUST be the 32-byte value used to generate the request nonce value from the previous response packet.

The value of "request" MUST be the transmitted request packet, including the "ROUGHTIM" header.

The value of "response" MUST be the received response packet, including the "ROUGHTIM" header.

The value of "publicKey" MUST be the long-term key that the server was expected to use for deriving the response signature.

When the client's list of servers has an associated URL for malfeasance reports, it SHOULD send a report whenever it has performed a measurement sequence in accordance with Section 8.2 and detected that at least one of the responses is inconsistent with causal ordering. Since the failure of a popular Roughtime server can cause numerous clients to send malfeasance reports at the same time, clients MUST use a reporting mechanism that avoids overloading the server receiving the reports. Clients SHOULD use exponential backoff for this purpose, with an initial and minimum retry interval of at least 10 seconds.

Clients MUST NOT send malfeasance reports in response to signature verification failures or any other protocol errors.

9. Security Considerations

9.1. Confidentiality

This protocol does not provide any confidentiality. Given the nature of timestamps, such impact is minor.

9.2. Integrity and Authenticity

The Roughtime protocol only provides integrity and authenticity protection for data contained in the SREP tag. Accordingly, new tags SHOULD be added to the SREP tag whenever possible.

9.3. Generating Private Keys

Although any random 256-bit string can be used as a private Ed25519 key, it has a high risk of being vulnerable to small-subgroup attacks and timing side-channel leaks. For this reason, all private keys used in Roughtime MUST be generated following the procedure described in Section 5.1.5 of RFC 8032 [RFC8032].

9.4. Private Key Compromise

The compromise of a PUBK's private key, even past MAXT, is a problem as the private key can be used to sign invalid times that are in the range MINT to MAXT, and thus violate the good-behavior guarantee of the server. To protect against this, it is necessary for clients to query multiple servers in accordance with the procedure described in Section 8.2.

9.5. Quantum Resistance

Since the only supported signature scheme, Ed25519, is not quantum resistant, the Roughtime version described in this memo will not survive the advent of quantum computers.

9.6. Maintaining Lists of Servers

The infrastructure and procedures for maintaining a list of trusted servers and adjudicating violations of the rules by servers is not discussed in this document and is essential for security.

9.7. Amplification Attacks

UDP protocols that send responses significantly larger than requests, such as NTP, have previously been leveraged for amplification attacks. To prevent Roughtime from being used for such attacks, servers MUST NOT send response packets larger than the request packets sent by clients.

10. Privacy Considerations

This protocol is designed to obscure all client identifiers. Servers necessarily have persistent long-term identities essential to enforcing correct behavior. Generating nonces in a nonrandom manner can cause leaks of private data or enable tracking of clients as they move between networks.

11. Operational Considerations

It is expected that clients identify a server by its long-term public key. In multi-tenancy environments, where multiple servers may be listening on the same IP or port space, the protocol is designed so that the client indicates which server it expects to respond. This is done with the SRV tag.

12. IANA Considerations

12.1. Service Name and Transport Protocol Port Number Registry

IANA is requested to allocate the following entry in the Service Name and Transport Protocol Port Number Registry:

Service Name: Roughtime

Transport Protocol: tcp,udp

Assignee: IESG <iesg@ietf.org>

Contact: IETF Chair <chair@ietf.org>

Description: Roughtime time synchronization

Reference: [[this memo]]

Port Number: [[TBD1]], selected by IANA from the User Port range

12.2. Roughtime Version Registry

IANA is requested to create a new registry entitled "Roughtime Version Registry". Entries shall have the following fields:

Version ID (REQUIRED): a 32-bit unsigned integer

Version name (REQUIRED): A short text string naming the version being identified.

Reference (REQUIRED): A reference to a relevant specification document.

The policy for allocation of new entries SHOULD be: IETF Review.

The initial contents of this registry shall be as follows:

Version ID	Version name	Reference
0x0	Reserved	[[this memo]]
0x1	Roughtime version 1	[[this memo]]
0x2-0x7fffffff	Unassigned	
0x80000000-0xffffffff	Reserved for Private or Experimental use	[[this memo]]

Table 1

12.3. Roughtime Tag Registry

IANA is requested to create a new registry entitled "Roughtime Tag Registry". Entries SHALL have the following fields:

Tag (REQUIRED): A 32-bit unsigned integer in hexadecimal format.

ASCII Representation (REQUIRED): The ASCII representation of the tag in accordance with Section 4.1.3 of this memo.

Reference (REQUIRED): A reference to a relevant specification document.

The policy for allocation of new entries in this registry SHOULD be: Specification Required.

The initial contents of this registry SHALL be as follows:

Tag	ASCII Representation	Reference
0x00474953	SIG	[[this memo]]
0x00524556	VER	[[this memo]]
0x00565253	SRV	[[this memo]]
0x434e4f4e	NONC	[[this memo]]
0x454c4544	DELE	[[this memo]]
0x45505954	TYPE	[[this memo]]
0x48544150	PATH	[[this memo]]
0x49444152	RADI	[[this memo]]
0x4b425550	PUBK	[[this memo]]
0x5044494d	MIDP	[[this memo]]
0x50455253	SREP	[[this memo]]
0x53524556	VERS	[[this memo]]
0x544e494d	MINT	[[this memo]]
0x544f4f52	ROOT	[[this memo]]
0x54524543	CERT	[[this memo]]
0x5458414d	MAXT	[[this memo]]
0x58444e49	INDX	[[this memo]]
0x5a5a5a5a	ZZZZ	[[this memo]]

Table 2

13. References

13.1. Normative References

[RFC1738] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, DOI 10.17487/RFC1738, December 1994, <https://www.rfc-editor.org/rfc/rfc1738>.

[RFC20] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/rfc/rfc20>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/rfc/rfc4086>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/rfc/rfc4291>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/rfc/rfc6234>.

[RFC6874] Carpenter, B., Cheshire, S., and R. Hinden, "Representing IPv6 Zone Identifiers in Address Literals and Uniform Resource Identifiers", RFC 6874, DOI 10.17487/RFC6874, February 2013, <https://www.rfc-editor.org/rfc/rfc6874>.

[RFC791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/rfc/rfc791>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[RFC9170] Thomson, M. and T. Pauly, "Long-Term Viability of Protocol Extension Mechanisms", RFC 9170, DOI 10.17487/RFC9170, December 2021, <https://www.rfc-editor.org/rfc/rfc9170>.

13.2. Informative References

[Merkle] Merkle, R. C., "A Digital Signature Based on a Conventional Encryption Function", in Pomerance, C. (eds) Advances in Cryptology, Lecture Notes in Computer Science vol 293, DOI 10.1007/3-540-47184-2_32, 1988, <https://doi.org/10.1007/3-540-48184-2_32>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <https://www.rfc-editor.org/rfc/rfc5905>.

[RFC738] Harrenstien, K., "Time server", RFC 738, DOI 10.17487/RFC0738, October 1977, <https://www.rfc-editor.org/rfc/rfc738>.

[RFC8915] Franke, D., Sibold, D., Teichel, K., Dansarie, M., and R. Sundblad, "Network Time Security for the Network Time Protocol", RFC 8915, DOI 10.17487/RFC8915, September 2020, <https://www.rfc-editor.org/rfc/rfc8915>.

Acknowledgments

Aanchal Malhotra and Adam Langley authored early drafts of this memo. Daniel Franke, Sarah Grant, Martin Langer, Ben Laurie, Peter Lthberg, Hal Murray, Tal Mizrahi, Ruben Nijveld, Christopher Patton, Thomas Peterson, Rich Salz, Dieter Sibold, Ragnar Sundblad, Kristof Teichel, Luke Valenta, David Venhoek, Ulrich Windl, and the other members of the NTP working group contributed comments and suggestions as well as pointed out errors.

Authors' Addresses

Watson Ladd
Akamai Technologies
Email: watsonbladd@gmail.com

Marcus Dansarie
Netnod
Email: marcus@dansarie.se
URI: https://orcid.org/0000-0001-9246-0263