B. Dowling
D. Stebila
Queensland University of Technology
G. Zaverucha
Microsoft Research
Draft, February 2015

ANTP: Authenticated NTP
Implementation Specification

Abstract

This document describes ANTP, an authentication protocol designed to be built over the Network Time Protocol operating in client/server mode. ANTP's design meets the requirements of NTP and the Security Requirements of Time Protocols in Packet-Switched Networks, a TICTOC Working Draft. In particular, the server does not need to keep per-client state, and the authentication steps does not degrade timestamp accuracy when compared to unauthenticated NTP. This specification is meant to accompany a paper describing ANTP and analyzing its security [ANTPpaper].

Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [RFC2119].

Table of Contents

1. Introduction 2
2. Overview 4
3. Protocol Flow Diagram 5
4. Processing Sequence 6
4.1. Negotiation Phase 6
4.2. Key-Exchange Phase 8
4.3. Time-Synchronization Phase 9
5. Protocol Messages 10
5.1. Negotiation Phase 12
5.1.1. Message: Client Negotiation 12
5.1.2. Message: Server Negotiation 13
5.2. The Key-Exchange Phase 14
5.2.1. Message: ClientKEX 14
5.2.2. Message: ServerKEX 15
5.3. Time Synchronization Phase 15
5.3.1. Message: ClientRequest 15
5.3.2. Message: ServerResponse 16
6. Security Model 17
7. Cryptographic Algorithms 18
7.1. Authenticated Encryption 18
7.2. Hash Algorithms 19
7.3. Key-Exchange Algorithms 19
7.4. Key Derivation Function 20
7.5. Message Authentication Code scheme 20
8. Security Considerations 21
8.1. Client Key Renegotiation 21
9. References 21
9.1. Normative References 21
9.2. Informative References 22
Authors' Addresses 22

1. Introduction

Many systems have increasing dependency on digital certificates and public-key infrastructure in order to authenticate and ensure confidentiality of communications. The Transport Layer Security (TLS or SSL), Secure Shell (SSH) and Internet Protocol Security (IPSec) protocols all utilize certificates to this effect, but many of these certificate schemes (in particular, X.509 certificates) utilize periods of validity and revocation lists in order to confirm that the key-pair is valid at the time of use. Thus, certificates in turn have a dependency of the synchronization of time between the issuer and the verifier. A malicious party that has the ability to offset the verifier's system clock from the issuer's can replay previously revoked or compromised certificates.

Secure time-synchronization is therefore desirable, but how does a party authenticate another when the public-key infrastructure itself is dependent on receiving secure time? The most widely deployed protocol for network time synchronization is the Network Time Protocol, which combats this problem by querying multiple parties and combining the recieved time-samples using Byzantine Agreement mechanisms to find a majority offset to the local clock. However, this assumes that most implementations:

a. Query multiple parties, which in typical deployments may not be the case (as evidenced by the implementation of the Simple Network Time Protocol)
b. Assumes the attacker can only affect some minority of the queried parties, which is not a realistic assumption for a man-in-the-middle attacker in control of the network.

This document specifies a protocol for the purpose of authenticating an NTP server to a client via public-key authentication. The assumption underpinning our construction is that the client has the ability to validate certificates in an out-of-band method. This is due to the circular dependency of time-synchronization and public-key authentication discussed earlier: without knowledge of the "correct" time, a party cannot validate a certificate, which is necessary to securely synchronize time. We discuss mechanisms for solving this problem in the Security Considerations section. Alternatively, the client time may be trusted to be sufficiently accurate for the purposes of certificate validation. For instance, if the client time is known to be offset by at most a week, this may be acceptable when verifying the validity of a certificate with a multi-year validity period.

The Network Time Security protocol [NTS] is an in-progress alternative security protocol that uses public-key infrastructure in order to secure time-synchronization protocols such as NTP and the Precise-Time Protocol (PTP). However, NTS is more costly in terms of

server-side public-key operations, vulnerable to downgrade attacks in the negotiation phase, and does not offer the client a mechanism to offset time-sample degradation caused by authentication of messages, but still achieve message authentication.

The Network Time Protocol version 4 also standardized a public-key-based authentication mechanism, but has multiple security vulnerabilities as examined in [Roettger].

A detailed survey of existing time syncrhonization protocols and security mechanisms is provide in a paper describing ANTP [ANTP paper]. The paper also provides a formal security analysis of ANTP, proving it is secure under standard assumptions about the underlying cryptographic primitives.

2. Overview

The basic design of this protocol is to construct a secure public-key authentication method over NTP. NTP builds a hierarchy where primary servers with direct access to reliable hardware clocks push synchronization to secondary servers that continue to push synchronization to further secondary servers and clients. A secondary time server may push synchronization to many clients, and this causes multiple problems when considering the possibility of constructing a secure time-synchronization protocol. First, creating a public-key authentication scheme on top of NTP may degrade the accuracy of the time-samples. NTP works by sending a timestamp as close as possible to the message transmission time in use for synchronization. Any kind of authentication operation on the timestamp itself creates an additional delay that adds offset between the receiver and sender of the timestamp. If it is a public-key operation, then this delay can be significant and variable. Second, requiring precise time-synchronization for a large number of clients demands a stateless protocol, precluding the use of a TLS-like authentication protocol. We thus have the following design goals:

1. Server-side processing of time-synchronization protocol messages requires no per-client state.
2. The client is capable of authenticating the server.
3. The client is capable of authenticating all messages from the server.
4. In one configuration, authentication adds no delay to time-samples when compared to unauthenticated NTP.
5. Authentication is optional, and ANTP clients can interoperate with deployed NTP servers (and vice-versa).
6. UDP packets are limited in size and the added necessity of transporting certificate chains means that messages may require fragmentation.

7. Replay attacks are explicitly prevented.

3. Protocol Flow Diagram

We begin by presenting a standard message flow for ANTP. There are three seperate phases to ANTP: The Negotiation Phase, the Key-Exchange Phase and the Time-Synchronization Phase. All messages are included in the extension field of an NTP packet (Version 4 or later). If an NTP server does support ANTP, the presence of an unknown extension will cause the packet to be ignored. If the client recieves an unauthenticated NTP response after any ANTP request, the client MUST abort the protocol and restart the protocol flow. This is to prevent an adversary from dropping the extension fields of an ANTP message in order to force downgrade attacks.

SNTP as standardized does not currently support extensions, but ANTP assumes SNTP behavior in order to compute and process time-synchronization messages and clock updates. Considering SNTP and NTP message structure is identical, we do not believe it would be difficult to also offer support for SNTP extensions.

When a client requires authentication, note that a response from a server that doesn't support ANTP is indistinguishable from a network attacker that has intercepted the client messages and sent an unauthenticated response. Thus an ANTP client MUST NOT pull synchronization from NTP in the event that an ANTP handshake fails: doing so would allow downgrade attacks, negating all security benefits of ANTP.

The Negotiation Phase exchanges lists of supported cryptographic algorithms, in order to negotiate a single protocol version, key-derivation function, hash function, key-exchange algorithm and MAC scheme for use in the protocol. It also allows the client to recieve the certificate of the server.

The Key-Exchange phase allows the two parties to negotiate a fresh shared secret key. It also authenticates the negotiation and key-exchange phases with the server providing a MAC of the negotiation and key-exchange messages utilizing the shared secret key.

The Time-Synchronization phase allows the client to recieve authenticated NTP messages utilizing the negotiated secret-key, hash algorithm and MAC scheme from the negotiation phase. The client can reuse the key in multiple consecutive time-synchronization phases, saving the server costly public-key operations. The time-synchronization packets also include a nonce to prevent replay attacks. Note also that the client MUST renegotiate the key periodically in order to maintain key-freshness.

The client request indicates whether the server should use a high accuracy response. For a high accuracy response, authentication is delayed. The server responds immediately with an unauthenticated response, then susequently sends the same response, but with authentication information. Servers MUST support the high accuracy option.

The ANTP protocol requires a setup stage from the server: For each key-exchange algorithm supported by the server, the server must generate long-term public and private parameters, and MUST obtain a X.509 digital certifcate issued by a trusted certificate authority over the public parameters. In addition, the server MUST choose an authenticated-encryption scheme, and a randomly generated a secret key K_s for use in the protocol.

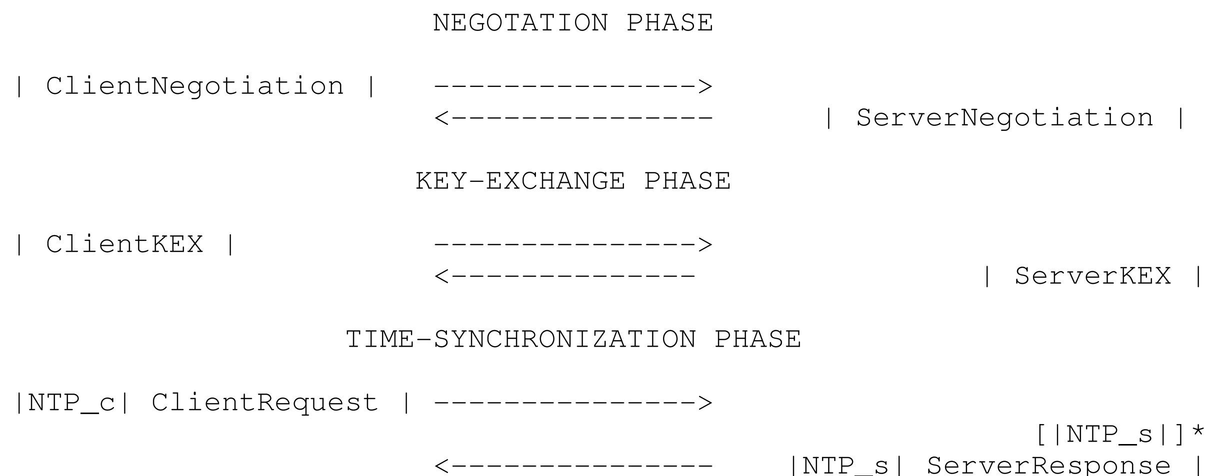

Figure 1. Protocol Flow for ANTP

* Indicates an optional message that is only sent if the client has requested a high accuracy response.

4. Processing Sequence

4.1. Negotiation Phase

1. The client begins by sending the ClientNegotiation message to the server, in the extension field of a correctly formatted NTP message. The client includes the client_kdf_algs, client_hash_algs, client_kex_algs, and client_mac_algs fields with the preferenced list of algorithms and the client_version with the highest supported

version of the client. The client also includes a 256-bit random nonce as a 32-byte array in the nonce field to ensure uniqueness of the message and defend against replay attacks. The client saves the ClientNegotiation message to authenticate the negotiation phase later in the protocol run.

2. Upon reciept of a ClientNegotiation message, the server creates a ServerNegotiation message, including the server_kdf_algs, server_hash_algs, server_kex_algs, and server_mac_algs (the preferenced list of algorithms that the server supports and the server_version with the highest supported version of the server). The server then computes the negotiated key-derivation function (denoted KDF), hash function (H), key-exchange (KEX), and Message Authentication Code (MAC) algorithms and the version by:

a. In the case of KDF, hash, key-exchange and MAC algorithms, the highest preferenced client algorithms that the server also supports and;
b. In the case of version, the highest-numbered version that both parties support.

The ServerNegotiation message includes the certificate (chain) associated with the negotiated key-exchange algorithm. The server uses H to compute a hash of the concatenation of ClientNegotiation and ServerNegotiation messages, and encrypts the output concatenated with the negotiated algorithms with the long-term secret symmetric-key K_s and the preferred authenticated-encryption scheme of the server as follows:

```
opaque1 = AUTH-ENC(K_s,
H(ClientNegotiation||ServerNegotiation)||KDF||Hash||KEX||MAC)
```

where the opaque1 field of the ServerNegotiation message is set to m zero bytes, where m is the length of opaque1 (the rest of the message is completed). The values KDF, Hash, KEX and MAC are the negotiated values, a single byte each, as described in Section 7 The value opaque1 added to the ServerNegotiation message and sent to the client.

3. Upon reciept of the ServerNegotiation message, the client computes the negotiated key-derivation, hash key-exchange, MAC algorithms and version (in the same way as the Server), and confirms that the certificate in the ServerNegotiation message supports the correct key-exchange algorithm. If it does not, the client MUST abort. The client saves the ServerNegotiation message in order to authenticate the protocol run in the key-exchange phase.

4.2. Key-Exchange Phase

1. The client generates an NTP message and a ClientKEX extension field, using the negotiated algorithm identifiers as input to the neg_kdf, neg_hash, neg_kex and neg_mac fields respectively, and the negotiated version as the neg_version. The value opaque1 is part of the ClientKeyExchange message.

Each KEX algorithm has three functions KEX_client, KEX_server and KEX, used by the client and server to create inputs for key exchange and to compute the shared secret. Details for allowed key exchange algorithms are given in Section 7.3.

The client extracts the public-key pk_s and key exchange parameters kex_params from the server certificate. Then the public and secret parts of the key exchange are computed:

```
(public_kex_mat, Z) = KEX_client(kex_params, pk_s)
```

Details of KEX_client are given in Section 7.3. The value public_kex_mat is sent to the server in the KeyExchangeMaterial field. The client also saves the shared secret Z.

2. Upon reciept of a ClientKEX message, the server verifies the integrity of the opaque value, decrypts and parses it as:

```
opaque1_d||KDF||Hash||KEX||MAC = AUTH-DEC(secret_s, opaque1)
```

If decryption fails, the server MUST abort. Otherwise, the server generates the secret key material Z using secret-key sk_s associated with pk_s as follows:

```
Z = KEX_server(kex_params, sk_s, public_kex_mat)
```

If KEX_server fails, the server MUST abort. The server uses Z to derive the key k:

```
k = KDF-H(Z, "time", "ANTP",L)
```

where "time" and "ANTP" are the ASCII encodings of the strings ANTP (0x414e5450) and time (0x74696d65), and L is the length of the desired key. Note that L will be the MAC key length for H. The inputs to the KDF are explained in further detail in Section 7.4. The server then encrypts k using K_s as a second opaque value:

```
opaque2 = AUTH-ENC(K_s, k||KDF||Hash||KEX||MAC)
```

The value opaque2 is part of the ServerKEX message. The values KDF, Hash, KEX and MAC are the negotiated values, a single byte each, as described in Section 7 Now all previous messages (e.g ClientNegotiation, ServerNegotiation, ServerKEX and ClientKEX messages) are authenticated:

```
mac_tag = MAC-H(k, opaque1_d||ClientKEX||ServerKEX)
```

The value mac_tag value is sent to the client in the ServerKEX message. Verifying the tag authenticates both the key-exchange and negotiation phases (as opaque1_d is a hash value over ClientNegotiation and ServerNegotiation messages).

3. Upon reciept of the ServerKEX message, the client derives the key:

```
k = KDF-H(Z, "time", "ANTP", L)
```

and the other values are as in the corresponding server key derivation step.

The client verifies the MAC tag in the ServerKEX message by computing Tag = MAC-H(k,H(ClientNegotiation||ServerNegotiation)||ClientKEX||ServerKEX). If Tag and mac_tag differ, the client MUST abort. Otherwise, the client accepts the negotiation and key-exchange phases and saves opaque2 and k for the time-synchronization phase.

4.3. Time-Synchronization Phase

1. The client begins the time-synchronization process by generating an NTP message according to the NTP specfication. The client also randomly generates a 256-bit nonce, and includes the nonce as a 32-byte array in the nonce field of the ClientRequest message. The nonce ensures uniqueness of the synchronization and prevents replay attacks. In addition, the negotiated version neg_version, the negotiated key-derivation (neg_kdf), hash (neg_hash), key exchange (neg_kex) and MAC (neg_mac) algorithms MAC are sent in the appropriate fields, and the opaque2 value from the ServerKEX message is sent in the opaque1 field. Finally, if the client requires accuracy similar to unauthenticated NTP, the AccuracyFlag field is set to 0x01. Otherwise the flag is set to 0x00 and the ClientRequest is sent to the server.

2. Upon reciept of the ClientRequest, the server creates an unauthenticated NTP response as specified in the NTP standard. If the ClientRequest's AccuracyFlag = 0x01 the server sends the unauthenticated NTP response immediately without a ServerResponse extension field. The server then generates a new secret key by decrypting and parsing the opaque2 value in the ClientRequest message as follows:

```
k||KDF||Hash||KEX||MAC = AUTH-DEC(K_s, opaque2)
```

and creating an authentication tag for both client and server NTP messages (denoted by NTP_c and NTP_s respectively):

```
mac_tag = MAC-H(k, NTP_c||ClientRequest||NTP_s||ServerResponse)
```

and inputting the mac_tag into the Tag field, sending the NTP_s||ServerResponse message to the client.

3. Upon reciept of an NTP response from the server, the client immediately processes the NTP packet. If the client indiciated the AccuracyFlag:

(i) the client MUST NOT pull synchronization from an unauthenticated NTP packet unless a ServerResponse attached to an identical NTP packet verifies correctly, and
(ii) if no ServerResponse is recieved, or no unauthenticated packet is recieved, the client MUST abort the protocol.

The client verifies the ServerResponse by:

```
Tag = MAC-H(k, NTP_c||ClientRequest||NTP_s||ServerResponse)
```

The client uses NTP_s for synchronization only if mac_tag = Tag. Otherwise the client MUST abort.

5. Protocol Messages

Recall that all messages are designed to be sent in the NTP extension fields similarly to the Autokey Protocol [RFC5906]. When the msg_type of the extension field equals 0x01, 0x02, 0x03, 0x04, or 0x05 the client MUST NOT use the information in the NTP message fields for synchronization. If the msg_type of the extension fields equal 0x01 or 0x03 the server MAY process the NTP message normally. When the namefield reads 0x06 or 0x05, the client and server MUST process the respective NTP messages as specified in the NTP specification.

This protocol follows DTLS [RFC4347] regarding message fragmentation. If the message requires fragmentation, the client divides the message into a series of N contiguous data ranges, each at least 56 bytes shorter than the maximum message size (to account for the NTP Packet and the msg_type, Length, Offset and FragmentLength field lengths). Each of these N data ranges becomes a new message, each attached to an identical NTP packet, and with identical msg_type and length. The Offset field of a message fragment is the number of bytes in previous fragments, and FragmentLength is the length of the current message fragment. When any party recieves an NTP message with an extension field containing a msg_type with value 0x01 (ClientNegotiation), 0x02 (ServerNegotiation), 0x03 (ClientKEX), 0x04 (ServerKEX), 0x05 (ClientRequest), 0x06 (ServerRequest), the party checks if length=FragmentLength. If not, the party MUST buffer until it has the entire message, and process as if the message were a single NTP packet attached to a extension field with a zeroed Length, FragmentLength, and FragmentOffset fields. This fragmentation strategy is applied to each ANTP protocol message, as required. Setting the maximum message length depends on the path MTU between the client and server. Clients can use path MTU discovery [RFC1191] [RFC1981]. See also Section 4.1.1.1 "PMTU Discovery" from [RFC4347] for information on how path MTU is set in DTLS.

We specify the following structure to describe the FragmentInfo structure utilized in all ANTP packets:

```
struct {
uint24 length;
uint16 offset;
uint16 FragmentLength;
} FragmentInfo
```

where:

length: An unsigned 24-bit integer describing the length of the unfragmented message

fragment_offset: An unsigned 16-bit integer describing the number of bytes contained in previous fragments of the message. When the message requires no fragmentation this value is 0.

fragment_length: An unsigned 16-bit integer describing the length of this fragment on the message. When the message requires no fragmentation, this value is length.

Note that since ANTP allows buffering of messages, it is possible that multiple ANTP messages that require fragmentation may be recieved by another party interleaved. Since each ANTP message that is fragmented is attached to an identical NTP message, it is trivial to distinguish fragmented ANTP messages via the NTP packet. In order

to reduce complexity however, the parties MUST NOT send multiple ANTP messages with identical NTP packets, but instead generate a new NTP message for each message flow.

In a similar way to TLS all values are stored in big-endian format, and the smallest block size is a single byte. We define variable-length vectors by specifying a range of legal lengths and sizes of the elements in the vector as follows:

```
type Name <floor,...,ceiling>
```

where type is the type of each element, floor is the smallest number of elements in the vector, and ceiling the largest. Note that for each vector the number of elements in the vector is prepended to the vector as an unsigned integer, using as many bytes as necessary to express ceiling (the length of the largest possible vector).

We define the following structure to represent a variable-length string of bytes:

```
struct {
uint32 length;
uint8 data<0, ..., 2^32 -1>
} ByteString
```

where:

length: An unsigned 32-bit integer indicating the number of bytes that follow.
data: A sequence of bytes (octets).

Note that for the ByteString structure, the data field is not serialized as a vector (with the length prepended), as the length is explicitly given by the first field.

5.1. Negotiation Phase

The negotiation phase begins with the exchange of messages to negotiate the key-exchange, hash algorithms and versions to be used throughout the protocol. In addition, the server sends the certificate necessary to validate the public-key of the server.

5.1.1. Message: Client Negotiation

The negotiation phase begins with the client sending the first negotiation message, with the following structure. The description of each field can be found below.

```
struct {
uint8 msg_type = 0x01;
FragmentInfo f;
uint8 client_version;
uint8 client_kdf_algs<1,...,255>;
uint8 client_hash_algs<1,...,255>;
uint8 client_kex_algs<1,...,255>;
uint8 client_mac_algs<1,...,255>;
uint8 nonce<0,...,31>;
} ClientNegotiation
```

msg_type A unsigned byte of value 0x01 indicating the ClientNegotiation message.

client_version: An unsigned 8-bit integer indicating the highest supported version of ANTP that the client supports.

client_kdf_algs: An ordered list of unsigned 8-bit integers representing the preferred key-derivation functions supported by the client.

client_hash_algs: An ordered list of unsigned 8-bit integers representing the preferred hash algorithms supported by the client.

client_kex_algs: An ordered list of unsigned 8-bit integers representing the preferred key-exchange algorithms supported by the client.

client_mac_algs: An ordered list of unsigned 8-bit integers representing the preferred MAC schemes supported by the client.

nonce: A random 256-bit value as a 32-byte array.

5.1.2. Message: Server Negotiation

The negotiation phase continues with the server processing the ClientNegotiation message and sending the ServerNegotiation message, with the following structure:

```
struct {
uint8 msg_type = 0x02;
FragmentInfo f;
uint8 server_version;
uint8 server_kdf_algs<1,...,255>;
uint8 server_hash_algs<1,...,255>;
uint8 server_kex_algs<1,...,255>;
uint8 server_mac_algs<1,...,255>;
ByteString server_cert;
ByteString opaque1
} ServerNegotiation
```

server_neg: A unsigned byte of value 0x02 indicating the ServerNegotiation message.

server_version: An unsigned 8-bit integer indicating the highest supported version of the authentication protocol that the server supports.

server_kdf_algs: An ordered list of unsigned 8-bit integers representing the preferred key-derivation functions supported by the server.

server_hash_algs: An ordered list of unsigned 8-bit integers representing the preferred hash algorithms supported by the server.

server_kex_algs: An ordered list of unsigned 8-bit integers representing the preferred key-exchange algorithms supported by the server.

server_mac_algs: An ordered list of unsigned 8-bit integers representing the preferred MAC schemes supported by the server.

server_cert: The certificate containing the server public-key. Note that the public-key corresponds to the key-exchange algorithm negotiated with the two ordered lists client_kex_algs and server_kex_algs.

opaque1 An encrypted value created by the server, opaque to the client.

5.2. The Key-Exchange Phase

The key-exchange phase establishes secret-key material, and implicitly authenticates both the key-exchange and negotiation phases to the client.

5.2.1. Message: ClientKEX

The key-exchange phase begins with the client sending the ClientKEX message, with the following structure and description:

```
struct {
uint8 msg_type = 0x03;
FragmentInfo f;
uint8 neg_version;
uint8 neg_kdf;
uint8 neg_hash;
uint8 neg_kex;
uint8 neg_mac;
ByteString opaque1
ByteString kex_mat
} ClientKEX
```

msg_type: A unsigned byte of value 0x03 indicating the ClientKEX message.

neg_version: unsigned 8-bit integer describing the negotiated version of the protocol that the parties will be using.

neg_kdf: An unsigned 8-bit integer describing the negotiated key-exchange algorithm that the protocol will be using.
neg_hash: An unsigned 8-bit integer describing the negotiated hash algorithm that the protocol will be using.
neg_kex: An unsigned 8-bit integer describing the negotiated key-exchange algorithm that the protocol will be using.
neg_mac: An unsigned 8-bit integer describing the negotiated key-exchange algorithm that the protocol will be using.
opaque1: The opaque value sent in the ServerNegotiation message.
kex_mat: The public key exchange material.

5.2.2. Message: ServerKEX

The server now processes the ClientKEX message to compute the shared secret key. The server then produces a second opaque encryption, this time of the key, and generates a MAC tag authenticating the Negotiation and Key-Exchange phases. The structure and description of the ServerKEX message is as follows:

```
struct {
uint8 msg_type = 0x04;
FragmentInfo f;
ByteString opaque2
ByteString mac_tag
} ServerKEX
```

msg_type: A unsigned byte of value 0x04 indicating the ServerKEX message.
opaque2: A second encrypted value created by the server, opaque to the client.
mac_tag: The MAC of the concatenated hash value and KEX messages using the agreed key. The length of the tag is known to both parties based on the negotiated hash function, and clients MUST check that the recieved mac_tag has the correct length.

5.3. Time Synchronization Phase

The Time-Synchronization Phase allows the client to request synchronization from a server that has previously been authenticated and established a shared secret key.

5.3.1. Message: ClientRequest

The Time Synchronization phase begins with the client computing the NTP packet as specified in the NTP standards, and additionally completing the ClientRequest extension as structured and described below:

```
struct {
uint8 msg_type = 0x05;
FragmentInfo f;
uint8 neg_version;
uint8 neg_kdf;
uint8 neg_hash;
uint8 neg_kex;
uint8 neg_mac;
uint8 nonce<0,...,31>;
ByteString opaque2
uint8 AccuracyFlag flag
} ClientRequest
```

msg_type: A unsigned byte of value 0x05 indicating the ClientRequest message.
neg_version: unsigned 8-bit integer describing the negotiated version of the protocol that the parties will be using.
neg_hash: An unsigned 8-bit integer describing the negotiated key-derivation function that the protocol will be using.
neg_hash: An unsigned 8-bit integer describing the negotiated hash algorithm that the protocol will be using.
neg_kex: An unsigned 8-bit integer describing the negotiated key-exchange algorithm that the protocol will be using.
neg_mac: An unsigned 8-bit integer describing the negotiated MAC scheme that the protocol will be using.
nonce: An unsigned random 256-bit value as a 32-byte array.
opaque2: The opaque value sent in the ServerKEX message.
flag: An unsigned 8-bit integer describing whether the client requires high accuracy. Legal values are 0x01 (the flag is set) or 0x00 (the flag is not set).

5.3.2. Message: ServerResponse

The server processes the Client NTP request as standardized, and computes the NTP response. If the AccuracyFlag in the ClientRequest is 0x01, the server immediately sends the message without a ServerResponse extension. Afterwards, the server computes the ServerResponse fields as described below, and attaches it as an extension to the previously computed NTP packet, sending the message to the client.

```
struct {
uint8 msg_type = 0x06;
FragmentInfo f;
ByteString mac_tag
} ServerResponse
```

msg_type: A unsigned byte of value 0x06 indicating the
ServerResponse message.

mac_tag: The MAC of the concatenated ClientRequest and
ServerResponse messages using the derived secret-key. The
length of the tag is known to both parties based on the
negotiated hash function, and clients MUST check that the
recieved mac_tag has the correct length.

6. Security Model

We consider security of the Authentication Protocol for Network Time Protocol in the presence of an attacker that has the ability to create, reorder, replay, modify or drop messages at will. We attempt to address the following concerns outlined in the Security Requirements TICTOC Working Group Informational:

1. Packet Manipulation
2. Authentication and Authorization of Sender
3. Spoofing Attacks
4. Replay Attacks
5. Performance, no degradation of accuracy of clock
6. Key Freshness

The packet-manipulation and authentication of sender is fulfilled via authentication of packets and public-key authentication respectively. Replay protection is addressed by the inclusion of a client-generated nonce with each time-synchronization. Accuracy of timestamps is addressed by limiting server-side public-key operations, and the ability of the client to choose a high-accuracy variant of time-synchronization. Key freshness is provided by requiring the client to renegotiate a new key periodically. We note that the Security Requirements Informational also outlines additional threats outside the scope of our protocol:

1. Packet Dropping
2. Packet Delay Manipulation
3. DoS Attacks
4. Crypto Performance Attacks
5. Time Protocol DoS
6. Fraud Time Source
7. Rogue Master Attacks

We define these as outside the scope of our protocol, as a MITM attacker can trivially perform DoS, packet-dropping and packet-delay manipulation attacks. Additionally, a MITM attacker can trivially overwhelm a server with Key Exchange messages generated with random values but a real opaque, forcing the server to perform public-key operations processing the key-exchange material. The Time Protocol

DoS and Fraud Time Source attacks are within the bounds of NTP, and processing NTP messages is outside the bounds of ANTP as a design goal.

The associated ANTP paper [ANTP paper] provides a formal security analysis of ANTP, proving it is secure under standard assumptions about the underlying cryptographic primitives.

7. Cryptographic Algorithms

7.1. Authenticated Encryption

This section discusses the functions AUTH-ENC and AUTH-DEC used in ANTP. No interoperability is required from the authenticated encryption algorithm, as the value is entirely opaque to the client. It is critical for security, as a malicious party who can decrypt client Alice's opaque2 value may masquerade as the server and create valid ServerResponse messages for Alice. Therefore, the server MUST use one of the following algorithms:

AES-GCM as specfied in [GCM]
AES-CCM as specficed in [RFC3610]
AES-CBC combined with HMAC-SHA, as specifed in [CBC-HMAC]

Note that AES-GCM requires an explicit counter that is never reused, and thus the server MUST generate a nonce for each new opaque value to be encrypted, and attach the nonce to the end of the encrypted opaque value.

The security level (determined by the key length and algorithm choice) SHOULD meet or exceed the security level of the negotiated hash function.

An additional consideration is key-lifetime. Each authenticated encryption algorithm has a maximum amount of data that can be encrypted with a key. To avoid exceeding this limit, servers SHOULD generate a new authenticated encryption key every two months (or sooner) and update the key. This also serves to force clients to renegotiate a new key, as the opaque value will no longer decrypt correctly. Additionally, a server SHOULD generate a new key whenever a new certificate is used.

Servers may also choose to graudally "phase in" use of a new key, since when the opaque2 value in the ClientRequest message is rejected, the client will restart the key negotiation phase. If a large number of clients simultaneously being key exchange the computational costs may overwhelm the server. Servers may include a small amount of metadata (like a key identifier) as part of the

opaque2 value to allow them to identify which key was used to create the opaque value. This allows the server to continue using the first key, while migrating clients over to the new key. After a fixed period of time (such as a day) the server should delete the first key.

7.2. Hash Algorithms

Following the direction set by [NTS], it is required that all parties MUST support SHA-256, MAY support SHA-384, SHOULD support SHA-512 and MUST NOT support SHA-1, MD5 or 'weaker' hash algorithms. Note that the length of the mac_tag field is dependent on the size of the output of the hash algorithm negotiated. Each Hash function represented in the HashAlgorithm/s fields is assigned an unsigned 8-bit ID for negotiation:

0x00: SHA-256
0x01: SHA-384
0x02: SHA-512

7.3. Key-Exchange Algorithms

We note that in our protocol, the key-exchange algorithms are required to provide both authentication and confideniality of the secret key material without the server using a signature algorithm. The RSA-OAEP option is an encryption-based key transport, i.e., the client chooses a random key and encrypts it with the server's public key. The ECDH (elliptic-curve Diffie-Hellman) option has the client generate an ephemeral public key, and the server's long term public key is used for key agreement. Note that forward secrecy is not required: if a server private key is revealed, previous time synchronizations cannot be affected. Future synchronization phases of previously agreed keys are vulnerable until the client re-negotiates a new key (and recieves a new server certificate). For both schemes, the parameters (kex_params) are forced by the server certificate; in the case of RSA-OAEP kex_params are the modulus size and in the case of ECDH kex_params are the curve parameters. The client may abort if the server parameters are deemed too weak.

For interoperability, implementations MUST support RSA-OAEP with modulus size >= 2048 and MUST not support smaller modulus sizes. ECDH MUST be supported with the named curve secp256r1 [SEC2]. Other curves MAY be supported, but they must provide 128-bits of security or above. Servers MUST implement the point validation steps before operating on recieved points (as specified for ECDH in [SEC1]).

Each Key-Exchange algorithm represented in the ANTP message fields is assigned an unsigned 8-bit ID for negotiation:

0x00: RSA-OAEP
0x01: ECDH

Recall that KEX_client takes as input (kex_params, pk_s), and outputs (public_kex_mat, Z).

For ECDH, kex_params are the curve parameters. The client generates an ephemeral key pair (public_kex_mat, r) with public point public_kex_mat and private value r. The shared secret is the x-coordinate of the point Z = ECDH(pk_s, r).
For RSA-OAEP, Z is a randomly chosen 512-bit value, kex_params are the modulus length and the hash function(s) used for OAEP encoding. The value public_kex_mat is the encryption of Z with the public key pk_s.

Recall that KEX_server takes as input (kex_params, sk_s, public_kex_mat) and outputs Z.

For ECDH the server computes the shared secret as the x-coordinate of the point Z = ECDH(public_kex_mat, sk_s). If point validation on public_kex_mat fails, the server MUST abort.
For RSA-OAEP the server decrypts public_kex_mat with the private key sk_s and outputs the plaintext Z. If decryption fails the server MUST abort.

7.4. Key Derivation Function

We note that in the ANTP protocol, secure Key-Derivation Functions are required in order to securely generate a shared secret key from public-key materical. We define the KDF used in the key-exchange phase to be the KDF in counter mode defined in NIST SP 800-108 [SP800-108]. This construction uses HMAC as a PRF, which is HMAC-H, where H is the negotiated hash function. We use the notation: KDF-H(Z, context, label, L), where Z is the shared secret, the context and labels are additional inputs, and L is the output length. We additionally allow mechanisms for negotiation key-derivation functions similarly to hash and key-exchange algorithms. Each KDF represented in ANTP message fields is assigned an unsigned 8-bit ID for negotiation:

0x00: SP800-108

7.5. Message Authentication Code scheme

We note that in the ANTP protocol, secure MAC schemes are required in in order to authenticate the protocol run and thus the client's time-synchronization partner. HMAC is the default MAC scheme used in ANTP, but allow mechanisms for negotiating MAC schemes similarly to hash functions and key-exchange algorithms. When HMAC is negotiated, the hash function is the negotiated hash function. Each MAC scheme represented in the ANTP message fields is assigned an unsigned 8-bit ID for negotiation:

0x00: HMAC

8. Security Considerations

8.1. Client Key Renegotiation

Note that the client could reuse the same opaque value from the ServerKEX message until the Server generates a new authenticated encryption key K_s as described above. However, one of the goals outlined by the TICTOC Working Group Informational is key freshness. Following this, in ANTP the client SHOULD renegotiate a key by restarting the protocol flow every two days.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, September 2003.

[RFC5906] Haberman, B. and D. Mills, "Network Time Protocol Version 4: Autokey Specification", RFC 5906, June 2010.

[ANTPpaper] Dowling, B., Stebila, D., and G. Zaverucha, "Authenticated Network Time Synchronization", IACR ePrint Technical Report 2015/TBD, February 2015.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST SP 800-38D, November 2007.

[SP800-108]
Chen, L., "Recommendation for Key Derivation Using Pseudorandom Functions", NIST SP 800-108, October 2009.

[CBC-HMAC]
McGrew, D., Foley, J., and K. Patterson, "Authenticated Encryption with AES-CBC and HMAC-SHA", Internet Draft http://tools.ietf.org/html/draft-mcgrew-aead-aes-cbc-hmac-sha2-05, July 2014.

[SEC1] Standards for Efficient Cryptography, (SECG)., "SEC 1: Elliptic Curve Cryptography", Version 2.0, May 2009.

[SEC2] Standards for Efficient Cryptography, (SECG)., "SEC 2: Recommended Elliptic Curve Domain Parameters", Version 2.0, January 2007.

9.2. Informative References

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, April 2006.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, August 1996.

[NTS] Sibold, D., Roettger, S., and K. Teichel, "Network Time Security ", Internet Draft https://tools.ietf.org/html/draft-ietf-ntp-network-time-security-04, July 2014.

[Roettger]
Roettger, S., "Analysis of the NTP Autokey Protocol", , February 2012.

Authors' Addresses

Benjamin Dowling
Queensland University of Technology
b1.dowling@qut.edu.au

Douglas Stebila
Queensland University of Technology
stebila@qut.edu.au

Greg Zaverucha
Microsoft Research
gregz@microsoft.com