RTPSEC
Internet-Draft
Intended status: Standards Track
Expires: September 5, 2007

P. Zimmermann
Zfone Project
A. Johnston, Ed.
Avaya
J. Callas
PGP Corporation
March 4, 2007

ZRTP: Media Path Key Agreement for Secure RTP
draft-zimmermann-avt-zrtp-03

Status of this Memo

Copyright Notice

Abstract

This document defines ZRTP, a protocol for media path Diffie-Hellman exchange to agree on a session key and parameters for establishing Secure Real-time Transport Protocol (SRTP) sessions. The ZRTP protocol is media path keying because it is multiplexed on the same port as RTP and does not require support in the signaling protocol.

ZRTP does not assume a Public Key Infrastructure (PKI) infrastructure or require the complexity of certificates in end devices. For the media session, ZRTP provides confidentiality, protection against Man in the Middle (MITM) attacks, and, in cases where a secret is available from the signaling protocol, authentication. ZRTP can utilize two Session Description Protocol (SDP) attributes to provide discovery and authentication through the signaling channel. To provide best effort SRTP, ZRTP utilizes normal RTP/AVP profiles.

Table of Contents

1. Introduction

ZRTP is a key agreement protocol which performs Diffie-Hellman key exchange during call setup in the media path, and is transported over the same port as the Real-time Transport Protocol (RTP) [2] media stream which has been established using a signaling protocol such as Session Initiation Protocol (SIP) [17]. This generates a shared secret which is then used to generate keys and salt for a Secure RTP (SRTP) [3] session. ZRTP borrows ideas from PGPfone [13]. A reference implementation of ZRTP is available as Zfone [14].

The ZRTP protocol has some nice cryptographic features lacking in many other approaches to media session encryption. Although it uses a public key algorithm, it does not rely on a public key infrastructure (PKI). In fact, it does not use persistent public keys at all. It uses ephemeral Diffie-Hellman (DH) with hash commitment, and allows the detection of Man in the Middle (MITM) attacks by displaying a short authentication string for the users to read and compare over the phone. It has perfect forward secrecy, meaning the keys are destroyed at the end of the call, which precludes retroactively compromising the call by future disclosures of key material. But even if the users are too lazy to bother with short authentication strings, we still get reasonable authentication against a MITM attack, based on a form of key continuity. It does this by caching some key material to use in the next call, to be mixed in with the next call's DH shared secret, giving it key continuity properties analogous to SSH. All this is done without reliance on a PKI, key certification, trust models, certificate authorities, or key management complexity that bedevils the email encryption world. It also does not rely on SIP signaling for the key management, and in fact does not rely on any servers at all. It performs its key agreements and key management in a purely peer-to-peer manner over the RTP packet stream.

If the endpoints have a mechanism for knowing or retrieving the other endpoint's signature key, the short authentication string can be authenticated by exchanging a signature over the short authentication string.

ZRTP can be used and discovered without being declared or indicated in the signaling path. This provides the a best effort SRTP capability. Also, this reduces the complexity of implementations and minimizes interdependency between the signaling and media layers. When ZRTP is indicated in the signaling and the SDP attribute extensions are used, ZRTP has additional useful properties. When the signaling path has end-to-end integrity protection, the short authentication string can be compared automatically by the ZRTP endpoints. By sending a unique ZRTP Identifier (ZID) in the

signaling, ZRTP provides a useful binding between the signaling and media paths.

The following sections provide an overview of the ZRTP protocol, describe the key agreement algorithm and RTP message formats.

2. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119 and indicate requirement levels for compliant implementations [1].

3. Media Security Requirements

This section discuses how ZRTP meets all ten RTP security requirements discussed in Section 4 of [12].

Since ZRTP is a media path key agreement approach, it meets the following requirements:

R1: Forking and retargeting MUST work with all end-points being SRTP.

R2: Forking and retargeting MUST allow establishing SRTP or RTP with a mixture of SRTP- and RTP-capable targets.

R3: With forking, only the entity to which the call is finally established, MUST get hold of the media encryption keys.

Note: R4 is not present in [12].

R5: A solution SHOULD avoid clipping media before SDP answer without additional signalling.

ZRTP's use of Diffie-Hellman key agreement allows it to meet these requirements:

R6: A solution MUST provide protection against passive attacks.

R7: A solution MUST be able to support Perfect Forward Secrecy.

ZRTPs meet the following requirements with its handling of algorithm lists:

R8: A solution MUST support algorithm negotiation without incurring per-algorithm computational expense.

R9: A solution MUST support multiple cipher suites without additional computational expense.

The use of the a=zrtp-zid allows ZRTP to meet this requirement:

R10: Endpoint identification when forking.

The use of the optional signature block in the Confirm1 and Confirm2 messages allows ZRTP to meet this requirement:

R11: A solution MUST NOT require 3rd-party certs. If two parties share an auth infrastructure they should be able to use it.

4. Overview

This section provides a description of how ZRTP works. This description is non-normative in nature but is included to build understanding of the protocol.

ZRTP is negotiated the same way a conventional RTP session is negotiated in an offer/answer exchange using the standard AVP/RTP profile. The ZRTP protocol begins after two endpoints have utilized a signaling protocol such as SIP and are ready to send. If ICE [24] is being used, ZRTP begins after ICE has completed its connectivity checks.

ZRTP is multiplexed on the same ports as RTP. It uses a unique header that makes it clearly differentiable from RTP or STUN.

In environments in which sending ZRTP packets to non-ZRTP endpoints might cause problems and signaling path discovery is not an option, ZRTP endpoints can include the RTP header extension flag in normal RTP packets sent at the start of a session as a probe to discover if the other endpoint supports ZRTP. If the flag is received from the other endpoint, ZRTP messages can then be exchanged.

A ZRTP endpoint initiates the exchange by sending a ZRTP Hello message to the other endpoint. The purpose of the Hello message is to confirm the endpoint supports the protocol and to see what algorithms the two ZRTP endpoints have in common.

The Hello message contains the SRTP configuration options, and the ZID. Each instance of ZRTP has a unique 96-bit random ZRTP ID or ZID that is generated once at installation time. ZIDs are discovered during the Hello message exchange. The received ZID is used to look up retained shared secrets from previous ZRTP sessions with the endpoint.

A response to a ZRTP Hello message is a ZRTP HelloACK message. The HelloACK message simply acknowledges receipt of the Hello. Since RTP commonly uses best effort UDP transport, ZRTP has retransmission timers in case of lost datagrams. There are two timers, both with exponential backoff mechanisms. One timer is used for retransmissions of Hello messages and the other is used for retransmissions of all other messages after receipt of a HelloACK.

4.1. Key Agreement Modes

After both endpoints exchange Hello and HelloACK messages, the key agreement exchange can begin with the ZRTP Commit message. ZRTP supports a number of key agreement modes including both Diffie-Hellman and non-Diffie-Hellman modes as described in the following sections.

4.1.1. Diffie-Hellman Mode

An example ZRTP call flow is shown in Figure 1 below. Note that the order of the Hello/HelloACK exchanges in F1/F2 and F3/F4 may be reversed. That is, either Alice or Bob might send the first Hello message. Also, an endpoint that receives a Hello message and wishes to immediately begin the ZRTP key agreement can omit the HelloACK and send the Commit instead. In Figure 1, this would result in messages F2, F3, and F4 being omitted. Note that the endpoint which sends the Commit message is considered the initiator of the ZRTP session and drives the key agreement exchange. The Diffie-Hellman public values are exchanged in the DHPart1 and DHPart2 messages. SRTP keys and salts are then calculated.

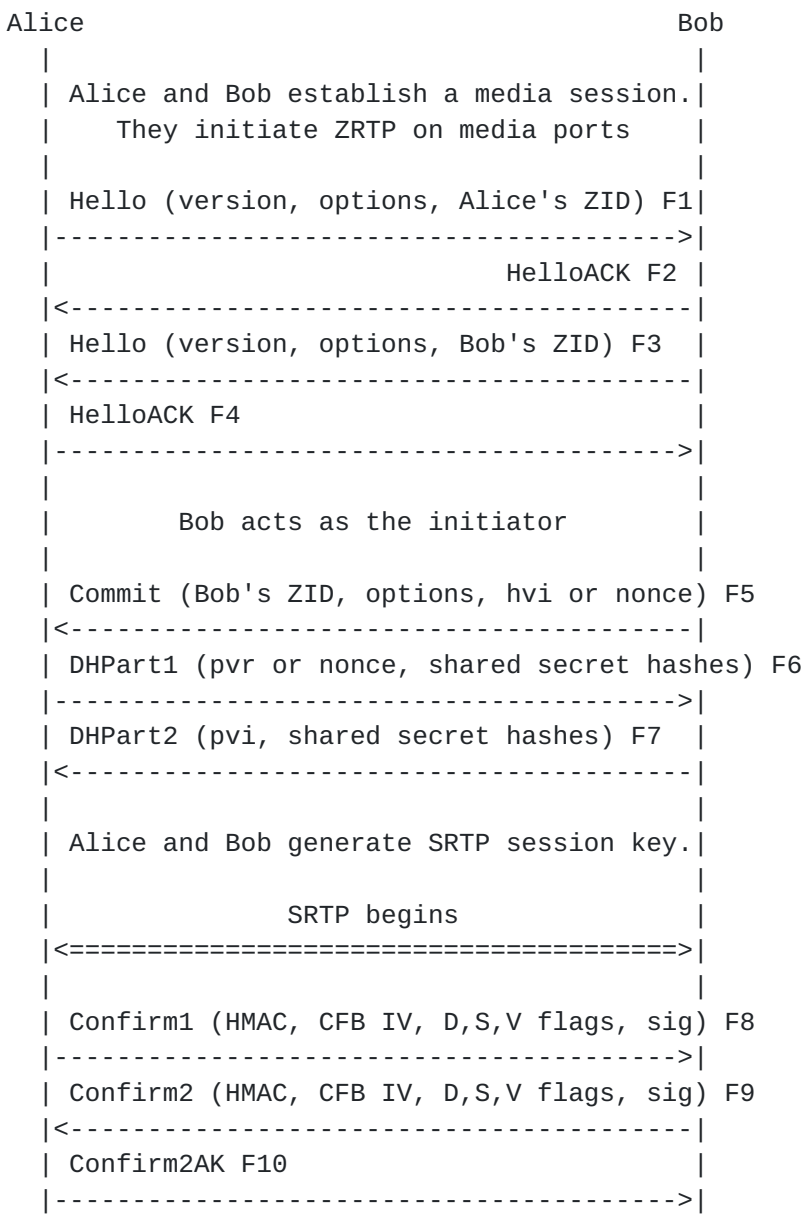

Figure 1. Establishment of an SRTP session using ZRTP

ZRTP authentication uses a Short Authentication String (SAS) which is ideally displayed for the human user. Alternatively, the SAS can be transported over the signaling channel in the SDP and compared automatically, provided the signaling has end-to-end integrity protection. Or, the SAS can be authenticated by exchanging a digital signature (sig) over the short authentication string in the Confirm1 or Confirm2 messages.

The ZRTP Confirm1 and Confirm2 messages are sent for a number of reasons. First, they confirm that all the key agreement calculations

were successful and thus the encryption will work, and they enable automatic detection of a DH MITM attack from a reckless attacker who does not know the retained shared secret. Digital signatures over the SAS can be exchanged to authenticate the exchange. And, they enable ZRTP to transmit some parameters under cover of CFB encryption, such as the Disclosure flag (D), the Allow Clear flag (A), and most importantly the SAS Verified flag (V SAS Verified flag (V), shielding it from a passive observer who would like to know if the human users are in the habit of diligently verifying the SAS.

4.1.2. Preshared Mode

In the Preshared Mode, endpoints can skip the DH calculation if they have a shared secret from a previous ZRTP session. Preshared mode is indicated in the Commit message and results in the same call flow as Figure 1. The DHPart1 and DHPart2 messages are exchanged so that the set of shared secrets can be determined, but the pvr and pvi are omitted and no DH calculation is performed. Instead nonces from the Commit and DHPart1 are exchanged and used along with the retained secrets to derive the key material. This mode could be useful for slow processor endpoints so that a DH calculation does not need to be performed every session. Or, this mode could be used to rapidly re-establish an earlier session that was recently torn down or interrupted without the need to perform another DH calculation. Since the cache is not affected during this mode, multiple Preshared mode exchanges can be processed at a time between two endpoints.

5. Protocol Description

ZRTP MUST be multiplexed on the same ports as the RTP media packets.

To support best effort encryption [12], ZRTP uses normal RTP/AVP profile (AVP) media lines in the initial offer/answer exchange. The ZRTP SDP attribute flag a=zrtp-id defined in Appendix A SHOULD be used in all offers and answers to indicate support for the ZRTP protocol. In subsequent offer/answer exchanges after a successful ZRTP exchange has resulted in an SRTP session, the Secure RTP/AVP (SAVP) profile MAY be used.

5.1. Discovery

During the ZRTP discovery phase, a ZRTP endpoint discovers if the other endpoint supports ZRTP and the supported algorithms and options. This information is transported in a Hello message.

ZRTP endpoints SHOULD include the SDP attribute a=zrtp-zid in offers and answers, as defined in Appendix A. ZRTP MAY use an RTP [2]

extension field as a flag to indicate support for the ZRTP protocol in RTP packets as described in Appendix D.

The Hello message includes the ZRTP version, hash, cipher, authentication method and tag length, key agreement type, and Short Authentication String (SAS) algorithms that are supported. In addition, each endpoint sends and discovers ZIDs. The received ZID is used to retrieve previous retained shared secrets, rs1 and rs2. If the endpoint has other secrets, then they are also collected. Details on how to derive the signaling secret, sigs, and SRTP secret, srtps, are in Appendix A.

Additional shared secrets can be defined and used as other_secret. If no secret of a given type is available, a random value is generated and used for that secret to ensure a mismatch in the hash comparisons in the DHPart1 and DHPart2 messages. This prevents an eavesdropper from knowing how many shared secrets are available between the endpoints.

A Hello message can be sent at any time, but is usually sent at the start of an RTP session to determine if the other endpoint supports ZRTP, and also if the SRTP implementations are compatible. A Hello message is retransmitted using timer T1 and an exponential backoff mechanism detailed in Section 7 until the receipt of a HelloACK message or a Commit message.

5.2. Commit Contention Resolution

After receiving a Hello message from the other endpoint, a Commit message can be sent to begin the ZRTP key exchange. The endpoint that sends the Commit is known as the initiator, while the receiver of the Commit is known as the responder.

If both sides send Commit messages initiating a secure session at the same time, the Commit message with the lowest hvi value is discarded and the other side is the initiator. This breaks the tie, allowing the protocol to proceed from this point with a clear definition of who is the initiator and who is the responder.

Because the DH exchange affects the state of the retained shared secret cache, only one in-process ZRTP DH exchange may occur at a time between two ZRTP endpoints. Otherwise, race conditions and cache integrity problems will result. When multiple media streams are established in parallel between the same pair of ZRTP endpoints (determined by the ZIDs in the Hello Messages), only one can be processed. Once that exchange completes with Confirm2 and Conf2ACK messages, another ZRTP DH exchange can begin. In the event that Commit messages are sent by both ZRTP endpoints at the same time, but

are received in different media streams, the same resolution rules apply - the Commit message with the lowest hvi value is discarded and the other side is the initiator. The media stream in which the Commit was sent will proceed through the ZRTP exchange while the media stream with the discarded Commit must wait for the completion of the other ZRTP exchange.

5.3. Shared Secret Determination

The following sections describe how ZRTP endpoints generate the set of shared secrets s1, s2, s3, s4, and s5 through the exchange of the DHPart1 and DHPart2 messages.

5.3.1. Responder Behavior

The responder calculates an HMAC keyed hash using the first retained shared secret, rs1, as the key on the string "Responder" which generates a retained secret ID, rs1IDr, which is truncated to 64 bits. HMACs are calculated in a similar way for additional shared secrets:

```
rs1IDr = HMAC(rs1, "Responder")

rs2IDr = HMAC(rs2, "Responder")

sigsIDr = HMAC(sigs, "Responder")

srtpsIDr = HMAC(srtps, "Responder")

other_secretIDr = HMAC(other_secret, "Responder")
```

The set of keyed hashes (HMACs) are included by the responder in the DHPart1 message.

The HMACs of the possible shared secrets received in the DHPart2 can be compared against the HMACs of the local set of possible shared secrets.

The expected HMAC values of the shared secrets are calculated (using the string "Initiator" instead of "Responder") as in Section 5.2.2 and compared to the HMACs received in the DHPart2 message. The secrets corresponding to matching HMACs are kept while the secrets corresponding to the non-matching ones are replaced with a null, which is assumed to have a zero length for the purposes of hashing them later. The set of up to five actual shared secrets are then s1, s2, s3, s4, and s5 - the order is that chosen by the initiator.

5.3.2. Initiator Behavior

The initiator calculates an HMAC keyed hash using the first retained shared secret, rs1, as the key on the string "Initiator" which generates a retained secret ID, rs1IDi, which is truncated to 64 bits. HMACs are calculated in a similar way for additional shared secrets:

rs1IDi = HMAC(rs1, "Initiator")

rs2IDi = HMAC(rs2, "Initiator")

sigsIDi = HMAC(sigs, "Initiator")

srtpsIDi = HMAC(srtps, "Initiator")

other_secretIDi = HMAC(other_secret, "Initiator")

These HMACs are included by the initiator in the DHPart2 message.

The initiator then calculates the set of secret IDs that are expected to be received from the responder in the DHPart1 message by substituting the string "Responder" instead of "Initiator" as in Section 5.3.1.

The HMACs of the possible shared secrets received are compared against the HMACs of the local set of possible shared secrets.

The secrets corresponding to matching HMACs are kept while the secrets corresponding to the non-matching ones are replaced with a null, which is assumed to have a zero length for the purposes of hashing them later. The set of up to five actual shared secrets are then s1, s2, s3, s4, and s5 - the order is that chosen by the initiator.

For example, consider two ZRTP endpoints who share secrets rs1, rs2, and a hash of a secret passphrase other_secret. During the comparison, rs1ID, rs2ID, and other_secretID will match but sigsID and srtpsID will not. As a result, s1 = rs1, s2 = rs2, s5 = other_secret, while s3 and s4 will be nulls.

5.4. Diffie-Hellman Mode

The purpose of the Diffie-Hellman exchange is for the two ZRTP endpoints to generate a new shared secret, s0. In addition, the endpoints discover if they have any shared secrets in common. If they do, this exchange allows them to discover how many and agree on an ordering for them: s1, s2, etc.

5.4.1. Hash Commitment

From the intersection of the algorithms in the sent and received Hello messages, the initiator chooses a hash, cipher, auth tag, key agreement type, and SAS type to be used.

A Diffie-Hellman mode is selected by setting the Key Agreement Type to DH4k or DH3k in the Commit. In this mode, the key agreement begins with the initiator choosing a fresh random Diffie-Hellman (DH) secret value (svi) based on the chosen key agreement type value, and computing the public value. (Note that to speed up processing, this computation can be done in advance.) For guidance on generating random numbers, see the section on Random Number Generation. The Diffie-Hellman secret value, svi, SHOULD be twice as long as the AES key length. This means, if AES 128 is used, the DH secret value SHOULD be 256 bits long. If AES 256 is used, the secret value SHOULD be 512 bits long.

pvi = g^svi mod p

where g and p are determined by the key agreement type value. The hash commitment is performed by the initiator of the ZRTP exchange. The hash value of the initiator, hvi, includes a hash of the Diffie-Hellman public value, pvi, and the responder's Hello message:

hvi=hash(pvi | responder's Hello message)

Note that the Hello message includes the fields shown in Figure 3.

The information from the responder's Hello message is included in the hash calculation to prevent a bid-down attack by modification of the responder's Hello message.

The initiator sends hvi in the Commit message.

5.4.2. Responder Behavior

Upon receipt of the Commit message, the responder generates its own fresh random DH secret value, svr, and computes the public value. (Note that to speed up processing, this computation can be done in advance.) For guidance on random number generation, see the section on Random Number Generation. The Diffie-Hellman secret value, svr, SHOULD be twice as long as the AES key length. This means, if AES 128 is used, the DH secret value SHOULD be 256 bits long. If AES 256 is used, the secret value SHOULD be 512 bits long.

pvr = g^svr mod p

Upon receipt of the DHPart2 message, the responder checks that the initiator's public DH value is not equal to 1 or p-1. An attacker might inject a false DHPart2 packet with a value of 1 or p-1 for g^svi mod p, which would cause a disastrously weak final DH result to be computed. If pvi is 1 or p-1, the user should be alerted of the attack and the protocol exchange must be terminated. Otherwise, the responder computes its own value for the hash commitment using the public DH value (pvi) received in the DHPart2 packet and its Hello packet and compares the result with the hvi received in the Commit packet. If they are different, a MITM attack is taking place and the user is alerted and the protocol exchange terminated.

The responder then calculates the Diffie-Hellman result:

DHResult = pvi^svr mod p

5.4.3. Initiator Behavior

Upon receipt of the DHPart1 message, the initiator checks that the responder's public DH value is not equal to 1 or p-1. An attacker might inject a false DHPart1 packet with a value of 1 or p-1 for g^svr mod p, which would cause a disastrously weak final DH result to be computed. If pvr is 1 or p-1, the user should be alerted of the attack and the protocol exchange must be terminated.

The initiator then sends a DHPart2 message containing the initiator's public DH value and the set of calculated retained secret IDs as described in 5.2.2.

The initiator calculates the same Diffie-Hellman result using:

DHResult = pvr^svi mod p

5.4.4. Shared Secret Calculation

The responder and initiator calculate the Diffie-Hellman shared secret:

DHSS = hash(DHResult)

A hash of the received and sent ZRTP messages in the current ZRTP exchange in the following order is calculated:

message_hash = hash (Hello of responder | Commit | DHPart1 | DHPart2)

Note that only the ZRTP message (Figures 3, 5, 6, and 7), not the entire ZRTP packets are included in the hash.

The final shared secret, s0, is calculated by hashing the concatenation of the DHSS and the set of non-null shared secrets as described in 5.2 and the message hash. As a result, the null secrets have no effect on the concatenation operation:

s0 = hash(DHSS | s1 | s2 | s3 | s4 | s5 | message_hash)

A new rs1 is calculated from s0:

rs1 = HMAC (s0, "retained secret")

After a successful exchange of Confirm1 and Confirm2 messaged described in Section 5.6, both sides now discard the rs2 value and store rs1 as rs2.

5.5. Preshared Mode

The Preshared key agreement mode can be used to generate SRTP keys and salts without a DH calculation, instead relying on one or more shared secrets from previous DH calculations between the endpoints.

This key agreement mode is useful for efficiently adding another media stream to an existing secure session, such as adding video to a session that already has performed a DH key agreement for the audio stream. It can also be used to rapidly re-establish a secure session between two parties who have recently started and ended a secure session that has already performed a DH key agreement, without performing another lengthy DH calculation, which may be desirable on slow processors in resource-limited environments.

5.5.1. Commit

This mode is selected by setting the Key Agreement Type to Preshared in the Commit message. From the intersection of the algorithms in the sent and received Hello messages, the initiator chooses a hash, cipher, auth tag, key agreement type, and SAS type to be used. In place of hvi in the Commit, a random number, nonce, 32 octets long is chosen. Its value MUST be unique for all nonce values chosen for all ZRTP sessions between a pair of endpoints since the last DH exchange. If a Commit is received with a reused nonce value, the ZRTP exchange SHOULD be immediately terminated. (We would say MUST be terminated, but we recognize it may be hard to determine if the nonce was never used before. In practical terms, a random nonce of this length has effectively no chance of repeating by accident.)

Note: Since nonces are used to calculate different SRTP key and salt pairs for each media session, a reuse of a nonce may result in the same key and salt being generated for multiple streams which would

introduce a major security weakness.

The DHPart1 and DHPart2 messages are exchanged in this mode so that the shared secrets can be determined. If it is determined that the endpoints have no shared DH secrets (i.e. either rs1 or rs2) the exchange MUST be terminated. It is RECOMMENDED that Preshared mode only be used when the SAS Verified flag is set.

5.5.2. Responder Behavior

In in place of pvr in the DHPart1, a random number, noncer, 32 octets long is chosen. Its value MUST be unique for all nonce values chosen for all ZRTP sessions between a pair of endpoints since the last DH exchange. If a DHPart1 is received with a reused nonce value, the ZRTP exchange SHOULD be immediately terminated. (We would say MUST be terminated, but we recognize it may be hard to determine if the nonce was never used before. In practical terms, a random nonce of this length has effectively no chance of repeating by accident.)

5.5.3. Initiator Behavior

Since no DH calculation is performed, no pvr is sent in the DHPart2 messages.

5.5.4. Shared Secret Calculation

A hash of the received and sent ZRTP messages in the current ZRTP exchange in the following order is calculated:

message_hash = hash (Hello of responder | Commit | DHPart1 | DHPart2)

Note that only the ZRTP message (Figures 3, 5, 6, and 7), not the entire ZRTP packets are included in the hash.

The final shared secret, s0, is calculated by hashing the concatenation of the set of non-null shared secrets as described in 5.3, and the message_hash.

s0 = hash(s1 | s2 | s3 | s4 | s5 | message_hash)

The noncei and noncer are implicitly included in the hash because they were included in the message hash.

No new retained shared secret is derived, and the values of rs1 and rs2 are unchanged during this mode.

5.6. Key Generation

The SRTP master key and master salt are then generated using the shared secret. Separate SRTP keys and salts are used in each direction for each media stream. Unless otherwise specified, ZRTP uses SRTP with no MKI, 32 bit authentication using HMAC-SHA1, AES-CM 128 or 256 bit key length, 112 bit session salt key length, 2^48 key derivation rate, and SRTP prefix length 0.

The ZRTP initiator encrypts and the ZRTP responder decrypts packets by using srtpkeyi and srtpsalti, which are generated by:

srtpkeyi = HMAC(s0,"Initiator SRTP master key")

srtpsalti = HMAC(s0,"Initiator SRTP master salt")

The key and salt values are truncated to the length determined by the chosen SRTP algorithm. The ZRTP responder encrypts and the ZRTP initiator decrypts packets by using srtpkeyr and srtpsaltr, which are generated by:

srtpkeyr = HMAC(s0,"Responder SRTP master key")

srtpsaltr = HMAC(s0,"Responder SRTP master salt")

The HMAC keys are generated by:

hmackeyi = HMAC(s0,"Initiator HMAC key")

hmackeyr = HMAC(s0,"Responder HMAC key")

Note that these HMAC keys are used only by ZRTP and not by SRTP.

Note: Different HMAC keys are needed for the initiator and the responder to ensure that GoClear messages in each direction are unique and can not be cached by an attacker and reflected back to the endpoint.

ZRTP keys are generated for the initiator and responder to use to encrypt the Confirm1 and Confirm2 messages.

zrtpkeyi = HMAC(s0,"Initiator ZRTP key")

srtpkeyr = HMAC(s0,"Responder ZRTP key")

The Short Authentication String (SAS) value is calculated as the hash of the ZRTP messages exchanged during the session: Hello from the responder, Commit, DHPart1, and DHPart2:

sasvalue = last 64 bits of message_hash

Note: The SAS calculated this way provides both protection against a bid down attack (modification of the Hello messages) or an active MiTM attack. Either attack will result in each endpoint calculating different sasvalues.

5.7. Confirmation

The Confirm1 and Confirm2 messages contain the cache expiration interval for the newly generated retained shared secret. The flagoctet is an 8 bit unsigned integer made up of the Disclosure flag (D), Allow clear flag (A), SAS Verified flag (V):

flagoctet = V * 2^2 + A * 2^1 + D * 2^0

Part of the Confirm1 and Confirm2 messages are encrypted using full-block Cipher Feedback Mode, and contain a 128-bit random CFB Initialization Vector (IV). The Confirm1 and Confirm2 messages also contain an HMAC covering the encrypted part of the Confirm1 or Confirm2 message which includes a string of zeros, the signature length, flag octet, cache expiration interval, signature type block (if present) and signature block (if present). For the responder

hmac = HMAC(hmackeyr, encrypted part of Confirm1)

For the initiator:

hmac = HMAC(hmackeyi, encrypted part of Confirm2 message)

The Conf2ACK message sent by the responder completes the exchange.

5.8. Random Number Generation

The ZRTP protocol uses random numbers for cryptographic key material, notably for the DH secret exponents and nonces, which must be freshly generated with each session. Whenever a random number is needed, all of the following criteria must be satisfied:

It MUST be derived from a physical entropy source, such as RF noise, acoustic noise, thermal noise, high resolution timings of environmental events, or other unpredictable physical sources of entropy. Chapter 10 of [7] gives a detailed explanation of cryptographic grade random numbers and provides guidance for collecting suitable entropy. The raw entropy must be distilled and processed through a deterministic random bit generator (DRBG). Examples of DRBGs may be found in NIST SP 800-90 [8], and in [7].

It MUST be freshly generated, meaning that it must not have been used in a previous calculation.

It MUST be greater than or equal to two, and less than or equal to 2^L - 1, where L is the number of random bits required.

It MUST be chosen with equal probability from the entire available number space, e.g., [2, 2^L - 1].

5.9. ZID and Cache Operation

Each instance of ZRTP has a unique 96-bit random ZRTP ID or ZID that is generated once at installation time. It is used to look up retained shared secrets in a local cache. A single global ZID for a single installation is the simplest way to implement ZIDs. However, it is specifically not precluded for an implementation to use multiple ZIDs, up to the limit of a separate one per callee. This then turns it into a long-lived "association ID" that does not apply to any other associations between a different pair of parties. It is a goal of this protocol to permit both options to interoperate freely.

Each time a new s0 is calculated, a new retained shared secret rs1 is generated and stored in the cache, indexed by the ZID of the other endpoint. The previous retained shared secret is then renamed rs2 and also stored in the cache. For the new retained shared secret, each endpoint chooses a cache expiration value which is an unsigned 32 bit integer of the number of seconds that this secret should be retained in the cache. The time interval is relative to when the Confirm1 message is sent or received.

The cache intervals are exchanged in the Confirm1 and Confirm2 messages. The actual cache interval used by both endpoints is the minimum of the values from the Confirm1 and Confirm2 messages. A value of 0 seconds means the secret should not be cached and the current values of rs1 and rs2 MUST be maintained. A value of 0xFFFFFFFF means the secret should be cached indefinitely and is the recommended value. If the ZRTP exchange results in no new shared secret generation (i.e. Preshared Mode), the field in the Confirm1 and Confirm2 is set to 0xFFFFFFFF and ignored, and the cache is not updated.

The expiration interval need not be used to force the deletion of a shared secret from the cache when the interval has expired. It just means the shared secret MAY be deleted from that cache at any point after the interval has expired without causing the other party to note it as an unexpected security event when the next key negotiation occurs between the same two parties. This means there need not be

perfectly synchronized deletion of expired secrets from the two caches, and makes it easy to avoid a race condition that might otherwise be caused by clock skew.

5.10. Terminating an SRTP Session or ZRTP Exchange

The GoClear message is used to switch from SRTP to RTP or to terminate an in-progress ZRTP exchange. The GoClear message contains a reason string for human purposes and a clear_hmac field.

When used to switch from SRTP to RTP, ZRTP uses an HMAC of the exact 4 octet Reason String sent in the GoGlear Message computed with the hmackey derived from the shared secret. When sent by the initiator:

clear_hmac = HMAC(hmackeyi, Reason String)

When sent by the responder:

clear_hmac = HMAC(hmackeyr, Reason String)

A GoClear message which does not receive a ClearACK response indicates that the GoClear has failed authentication (the clear_hmac does not validate) and that the session must stay in secure mode.

When terminating an in-progress ZRTP exchange, no secret hmackey is available, so the clear_hmac field is set to all zeros and ignored. The reason string SHOULD indicate the reason for the failure (e.g. "No Session Key", "Nonce Reuse", "Invalid DH Value"). The termination of a ZRTP key agreement exchange results in no updates to the cached shared secrets and deletion of all crypto context.

A ZRTP endpoint that receives a GoClear authenticates the message by checking the clear_hmac. If the message authenticates, the endpoint stops sending SRTP packets, generates a ClearACK in response, and deletes the crypto context for the SRTP session. Until confirmation from the user is received (e.g. clicking a button, pressing a DTMF key, etc.), the ZRTP endpoint MUST NOT resume sending RTP packets. The endpoint then renders the Reason String (after making sure only valid ASCII characters are present) and an indication that the media session has switched to clear mode to the user and waits for confirmation from the user. To prevent pinholes from closing or NAT bindings from expiring, the ClearACK message MAY be resent at regular intervals (e.g. every 5 seconds) while waiting for confirmation from the user. After confirmation of the notification is received from the user, the sending of RTP packets may begin.

After sending a GoClear message, the ZRTP endpoint stops sending SRTP packets. When a ClearACK is received, the ZRTP endpoint deletes the

crypto context for the SRTP session and may then resume sending RTP packets. However, the ZRTP Session key is not deleted unless the signaling session is terminated as well.

A ZRTP endpoint MAY choose to accept GoClear messages after the session has switched to SRTP, allowing the session to revert to RTP. This is indicated in the Confirm1 or Confirm2 messages by setting the Allow Clear flag (A). If the other endpoint set the Allow Clear (A) flag in their confirm message, GoClear messages MAY be sent after the session has gone secure.

Note: GoClear messages can always be sent prior to session going secure if the ZRTP exchange is terminated.

6. ZRTP Messages

All ZRTP messages use the message format defined in Figure 2. All word lengths referenced in this specification are 32 bits or 4 octets. All integer fields are carried in network byte order, that is, most significant byte (octet) first, commonly known as big-endian.

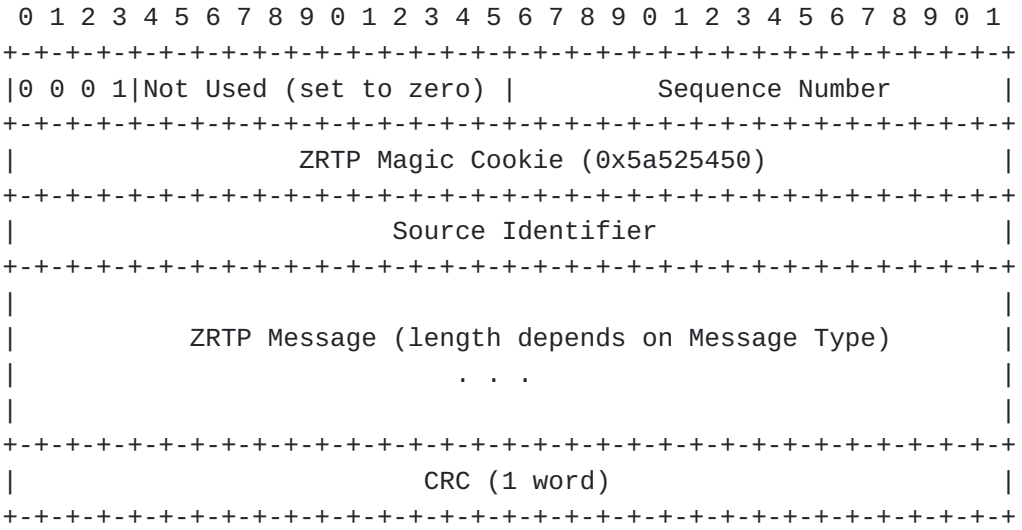

Figure 2. ZRTP Packet Format

The Sequence Number is a count that is incremented for each ZRTP packet sent. The count is initialized to a random value. This is useful in estimating ZRTP packet loss and also detecting when ZRTP packets arrive out of sequence.

The ZRTP Magic Cookie is a 32 bit string that uniquely identifies a ZRTP packet, and has the value 0x5a525450.

Source Identifier is the SSRC number of the RTP stream that this ZRTP packet relates to. For cases of forking or forwarding, RTP and hence ZRTP may arrive at the same port from several different sources - each of these sources will have a different SSRC and may initiate an independent ZRTP protocol session.

This format is clearly identifiable as non-RTP due to the first two bits being zero which looks like RTP version 0, which is not a valid RTP version number. It is clearly distinguishable from STUN since the magic cookies are different. The 12 not used bits are set to zero and MUST be ignored when received.

The ZRTP Messages are defined in Figures 3 to 11 and are of variable length.

The ZRTP protocol uses a 32 bit CRC checksum in each ZRTP packet as defined in RFC 3309 [6] to detect transmission errors. ZRTP packets are typically transported by UDP, which carries its own built-in 16-bit checksum for integrity, but ZRTP does not rely on it. This is because of the effect of an undetected transmission error in a ZRTP message. For example, an undetected error in the DH exchange could appear to be an active man-in-the-middle attack. The psychological effects of a false announcement of this by ZTRP clients can not be overstated. The probability of such a false alarm hinges on a mere 16-bit checksum that usually protects UDP packets, so more error detection is needed. For these reasons, this belt-and-suspenders approach is used to minimize the chance of a transmission error affecting the ZRTP key agreement.

The CRC is calculated across the entire ZRTP packet shown in Figure 2, including the ZRTP Header and the ZRTP Message, but not including the CRC field. If a ZRTP message fails the CRC check, it is silently discarded.

6.1. ZRTP Message Formats

ZRTP messages are designed to simplify endpoint parsing requirements and to reduce the opportunities for buffer overflow attacks (a good goal of any security extension should be to not introduce new attack vectors...)

ZRTP uses 8 octets (2 words) blocks to encode Message Type. 4 octets (1 word) blocks are used to encode Hash Type, Cipher Type, and Key Agreement Type, and Authentication Tag. The values in the blocks are ASCII strings which are extended with spaces (0x20) to make them the

desired length. Currently defined block values are listed in Tables 1-6 below.

Additional block values may be defined and used.

ZRTP uses this ASCII encoding to simplify debugging and make it "ethereal friendly".

6.1.1. Message Type Block

Currently ten Message Type Blocks are defined - they represent the set of ZRTP message primitives. ZRTP endpoints MUST support the Hello, HelloACK, Commit, DHPart1, DHPart2, Confirm1, Confirm2, Conf2ACK, GoClear and ClearACK block types.

Message Type Block	Meaning
"Hello "	Hello Message defined in Section 6.2
"HelloACK"	HelloACK Message defined in Section 6.3
"Commit "	Commit Message defined in Section 6.4
"DHPart1 "	DHPart1 Message defined in Section 6.5
"DHPart2 "	DHPart2 Message defined in Section 6.6
"Confirm1"	Confirm1 Message defined in Section 6.7
"Confirm2"	Confirm2 Message defined in Section 6.8
"Conf2ACK"	Conf2ACK Message defined in Section 6.9
"GoClear "	GoClear Message defined in Section 6.10
"ClearACK"	ClearACK Message defined in Section 6.11

Table 1. Message Block Type Values

6.1.2. Hash Type Block

Only one Hash Type is currently defined, SHA256, and all ZRTP endpoints MUST support this hash. Additional Hash Types can be registered and used.

Hash Type Block	Meaning
"S256"	SHA-256 Hash defined in [SHA-256]

Table 2. Hash Block Type Values

6.1.3. Cipher Type Block

All ZRTP endpoints MUST support AES128 and MAY support AES256 [4]. or other Cipher Types. Also, if AES 128 is used, DH3k should be used. If AES 256 is used, DH4k should be used.

Note: DH4k may be deprecated in the future in favor of elliptic curve algorithms.

Cipher Type Block	Meaning
"AES1"	AES-CM with 128 bit keys as defined in RFC 3711
"AES2"	AES-CM with 256 bit keys as defined in RFC 3711

Table 3. Cipher Block Type Values

6.1.4. Auth Tag Block

All ZRTP endpoints MUST support HMAC-SHA1 authentication, 32 bit and 80 bit length tags as defined in RFC 3711.

Auth Tag Block	Meaning
"HS32"	HMAC-SHA1 32 bit authentication tag as defined in RFC 3711
"HS80"	HMAC-SHA1 80 bit authentication tag as defined in RFC 3711

Table 4. Auth Tag Values

6.1.5. Key Agreement Type Block

All ZRTP endpoints MUST support DH3k and MAY support DH4k. ZRTP endpoints MUST use the DH generator function g=2. The choice of AES key length is coupled to the choice of key agreement type. If AES 128 is chosen, DH3k SHOULD be used. If AES 256 is chosen, DH4k SHOULD be used. ZRTP also defines a non-DH mode, Preshared, which SHOULD be supported. In Preshared mode, the SRTP key is derived from the set of shared secrets and a pair of nonces.

Note: DH4k may be deprecated in the future in favor of elliptic curve algorithms.

Key Agreement Type Block	Meaning
"DH3k"	DH mode with p=3072 bit prime as defined in RFC 3526
"DH4k"	DH mode with p=4096 bit prime as defined in RFC 3526
"Prsh"	Preshared Non-DH mode uses shared secrets.

Table 5. Key Agreement Block Type Values

6.1.6. SAS Type Block

All ZRTP endpoints MUST support the base32 and MAY support base256 Short Authentication String scheme, and other SAS rendering schemes. The ZRTP SAS is described in Section 7.

SAS Type Block	Meaning
"B32 "	Short Authentication String using base32 encoding defined in Section 8.
"B256"	Short Authentication String using base256 encoding defined in Section 8.

Table 6. SAS Block Type Values

The SAS Type determines how the SAS is rendered to the user so that the user may compare it with his partner over the voice channel. This allows detection of a man-in-the-middle (MITM) attack.

6.1.7. Signature Block

The signature type block is a 4 octet (1 word) block used to represent the signature algorithm. Suggested signature algorithms and key lengths are a future subject of standardization.

6.2. Hello message

The Hello message has the format shown in Figure 3. The Hello ZRTP message begins with the preamble value 0x505a then a 16 bit length in 32 bit words. This length includes only the ZRTP message (including the preamble and the length) but not the ZRTP header or CRC. Next is the Message Type Block and a 4 character string containing the version (ver) of ZRTP, currently "0.05". Next is the Client Identifier string (cid) which is 3 words long and identifies the vendor and release of the ZRTP software. The next parameter is the ZID, the 96 bit long unique identifier for the ZRTP endpoint. The next four bits contains flag bits. The only defined flag is the Passive bit (P), a Boolean normally set to False. A ZRTP endpoint which is configured to never initiate secure sessions is regarded as passive, and would set the P bit to True. The next 8 bits are unused. They should be set to zero when sent and ignored on receipt. Next is a list of supported Hash Types, Cipher Types, Auth Tag, Key Agreement Types, and SAS Type. The number of listed algorithms are listed for each type: hc=hash count, cc=cipher count, ac=auth tag count, kc=key agreement count, and sc=sas count. The values for these algorithms are defined in Tables 2, 3, 4, 5, and 6. A count of zero means that only the mandatory to implement algorithms are supported. Mandatory algorithms MAY be included in the list. The order of the list indicates the preferences of the endpoint. If a mandatory algorithm is not included in the list, it is added to the end of the list for preference.

Note: Implementers are encouraged to keep these algorithm lists small - the list does not need to include every cipher and hash supported, just the ones the endpoint would prefer to use for this ZRTP exchange.

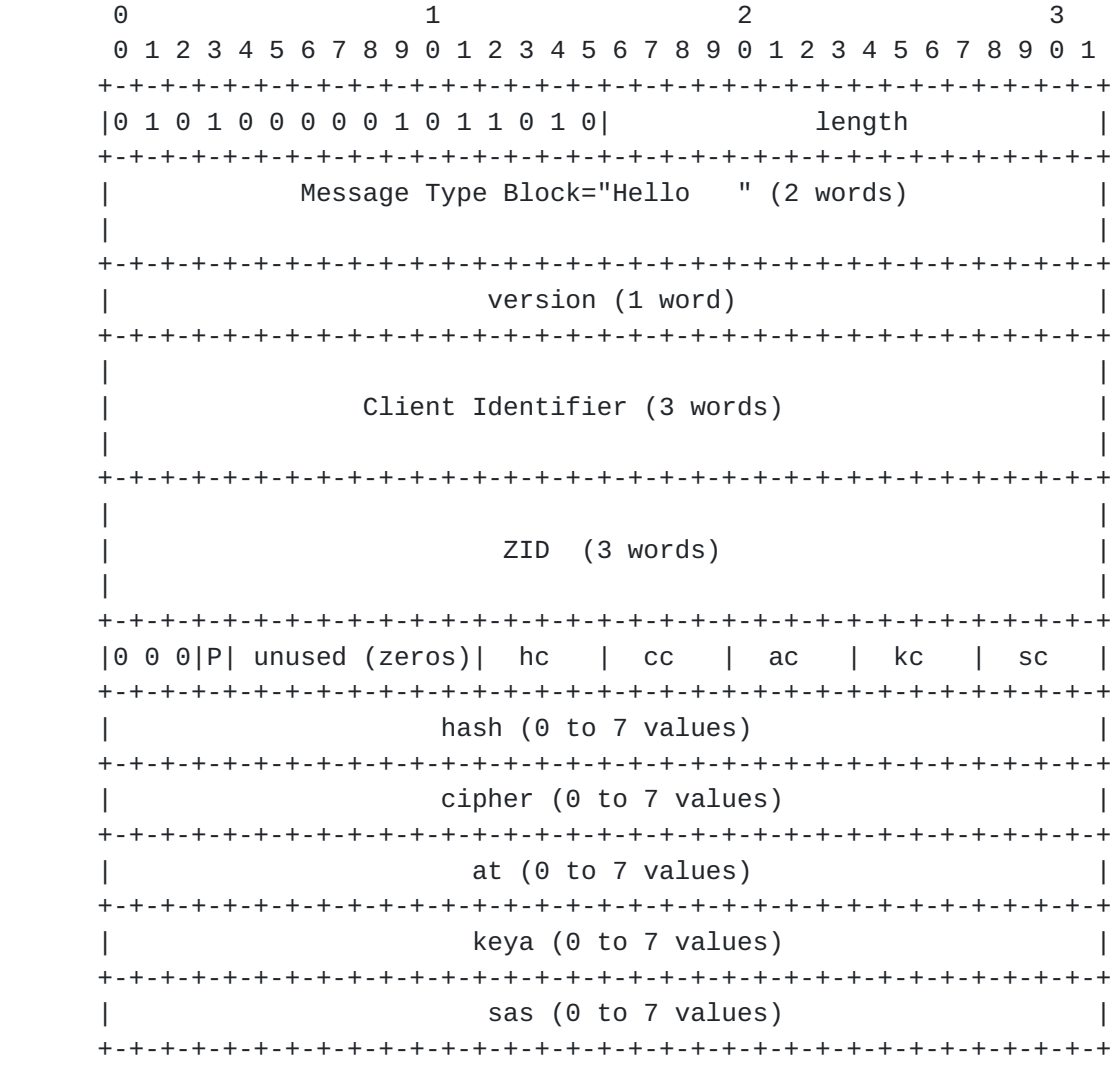

Figure 3. Hello message format

6.3. HelloACK message

The HelloACK message is used to stop retransmissions of a Hello message. A HelloACK is sent regardless if the version number in the Hello is supported or the algorithm list supported. The receipt of a HelloACK stops retransmission of the Hello message. The format is shown in Figure 4 below. Note that a Commit message can be sent in place of a HelloACK by an initiator.

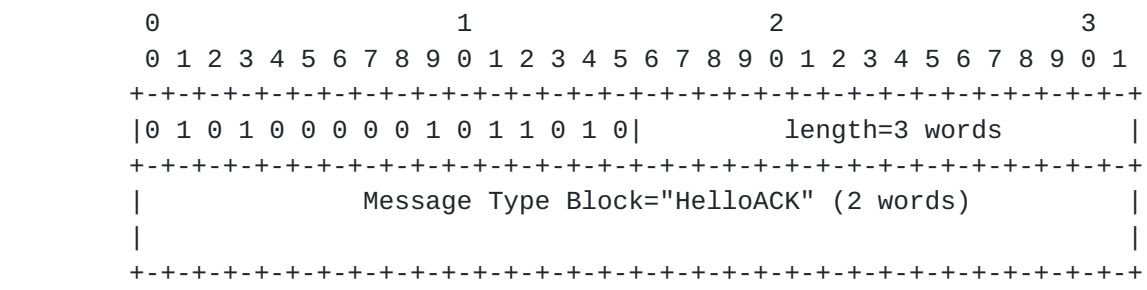

Figure 4. HelloACK message format

6.4. Commit message

The Commit message is sent to initiate the key agreement process after receiving a Hello message. The Commit message contains the initiator's ZID and a list of selected algorithms (hash, cipher, atl, keya, sas), the ZRTP mode, and hvi, a hash of the public DH value of the initiator and the algorithm list from the responder's Hello message. If a non-DH mode is used, hvi is replaced by a random number, noncei. The Commit Message format is shown in Figure 5.

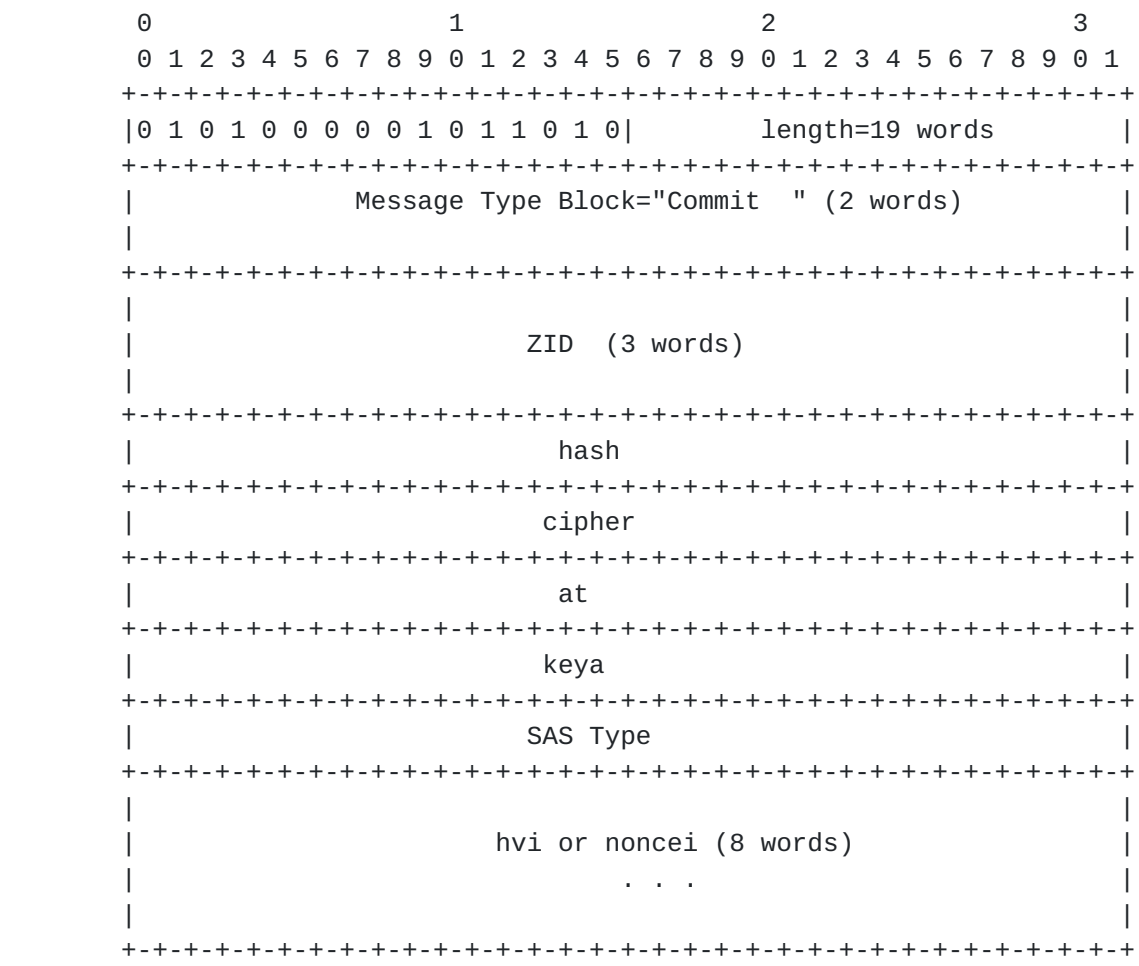

Figure 5. Commit message format

6.5. DHPart1 message

The DHPart1 message begins the DH exchange. The format is shown in Figure 5 below. The DHPart1 message is sent if a valid Commit message is received. The length of the pvr value depends on the Key Agreement Type chosen. If DH4k is used, the pvr will be 128 words (512 octets) and the length of this message will be 141 words. If DH3k is used, it is 96 words (384 octets) and the length of this message will be 109 words. If the Key Agreement Type is Preshared, then pvr is replaced by an 8 word noncer from the responder and the length of this message will be 21 words.

The next five parameters are HMACs of potential shared secrets used in generating the ZRTP secret. The first two, rs1IDr and rs2IDr, are the HMACs of the responder's two retained shared secrets, truncated to 64 bits. Next is sigsIDr, the HMAC of the responder's signaling secret, truncated to 64 bits. Next is srtpsIDr, the HMAC of the responder's SRTP secret, truncated to 64 bits. The last parameter is

the HMAC of an additional shared secret. For example, if multiple SRTP secrets are available or some other secret is used, it can be used as the other_secret. The Message format for the DHPart1 message is shown in Figure 6.

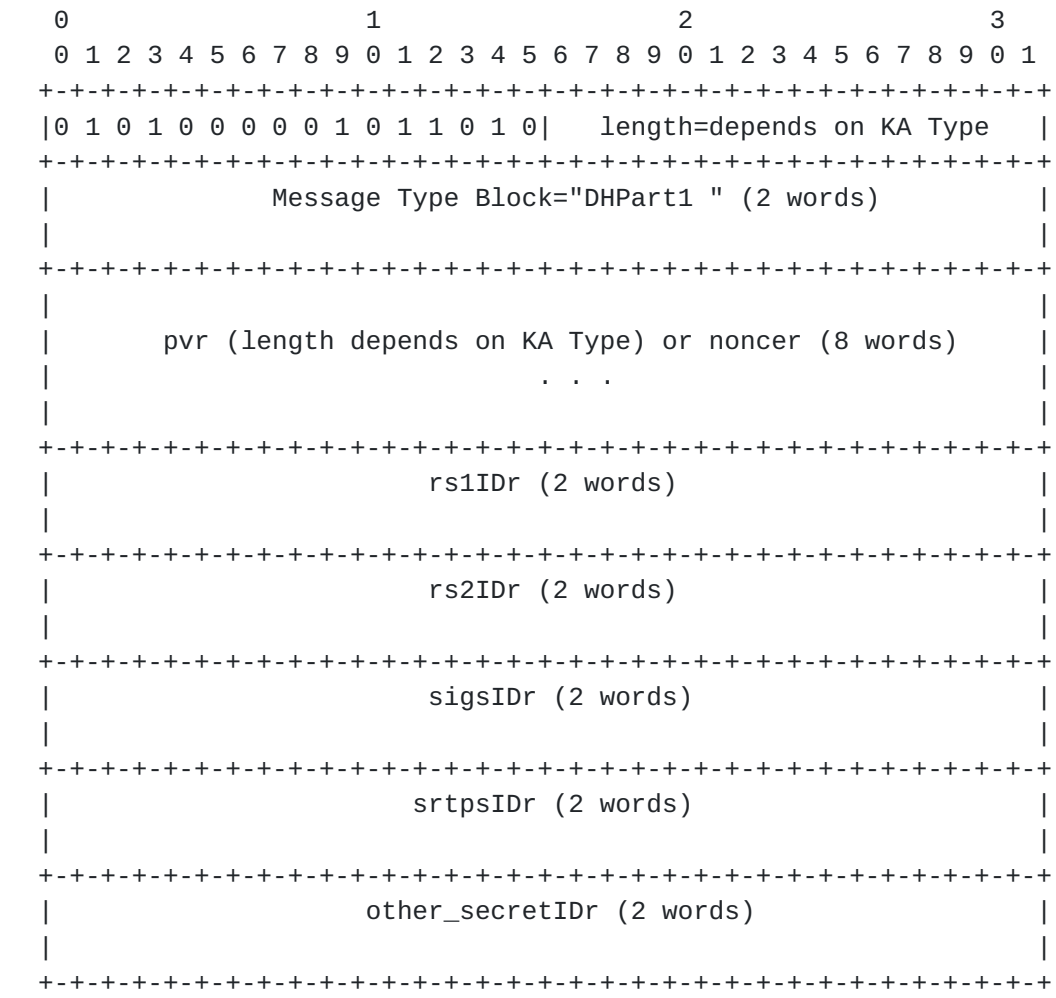

Figure 6. DHPart1 message format

6.6. DHPart2 message

The DHPart2 message completes the DH exchange. A DHPart2 message is sent if a valid DHPart1 message is received. The length of the pvi value depends on the Key Agreement Type chosen. If DH4k is used, the pvi will be 128 words (512 octets) and the length of this message will be 141 words. If DH3k is used, it is 96 words (384 octets) and the length of this message will be 109 words. If the Key Agreement Type is Preshared, then pvi is omitted (0 octets) and the length of this message will be 13 words.

The next five parameters are HMACs of potential shared secrets used in generating the ZRTP secret. The first two, rs1IDi and rs2IDi, are the HMACs of the initiator's two retained shared secrets, truncated to 64 bits. Next is sigsIDi, the HMAC of the initiator's signaling secret, truncated to 64 bits. Next is srtpsIDi, the HMAC of the initiator's SRTP secret, truncated to 64 bits. The last parameter is the HMAC of an additional shared secret. For example, if multiple SRTP secrets are available or some other secret is used, it can be included. The message format for the DHPart2 message is shown in Figure 7.

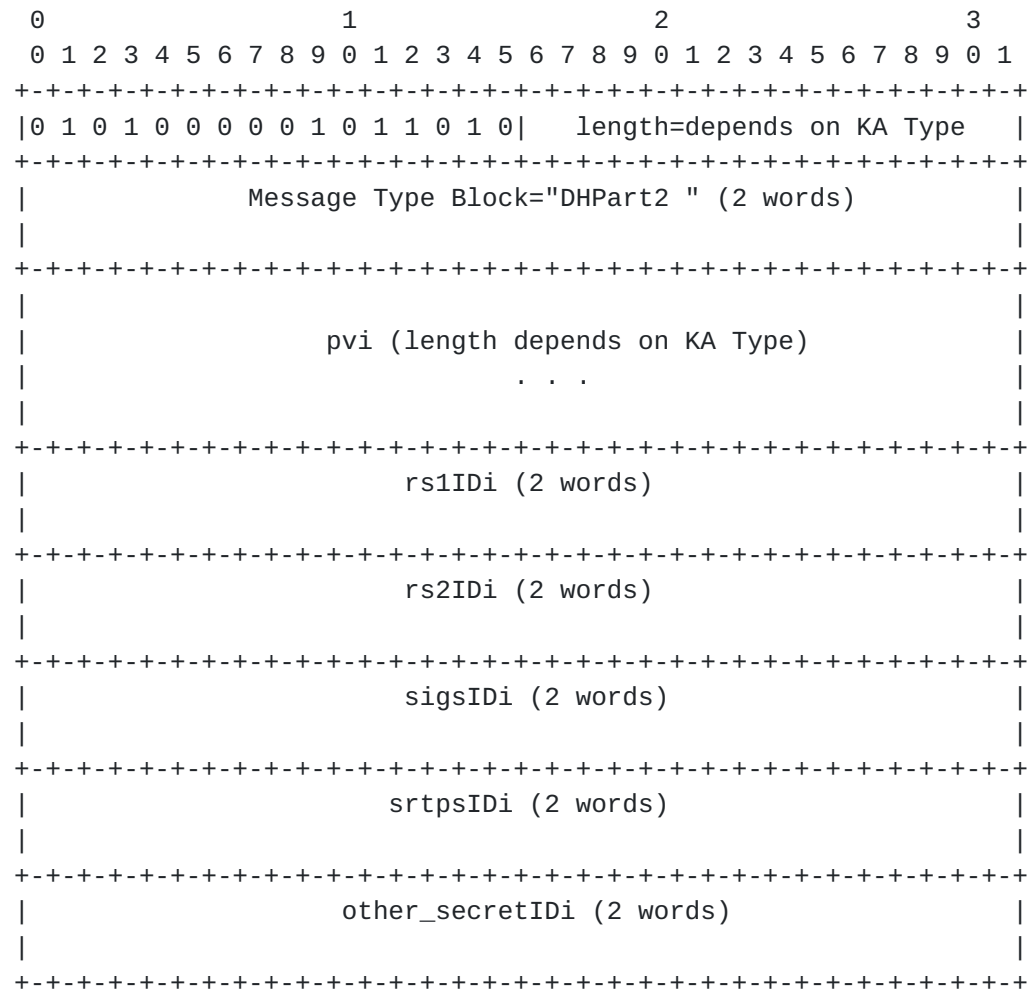

Figure 7. DHPart2 message format

6.7. Confirm1 and Confirm2 messages

The Confirm1 message is sent in response to a valid DHPart2 message after the SRTP session key and parameters have been negotiated. The Confirm2 message is sent in response to a Confirm1 message. The

format is shown in Figure 8 below. The message contains the Message Type Block "Confirm1" or "Confirm2". Next is the HMAC, a keyed hash over encrypted part of the message (shown enclosed by "===" in Figure 8.) The next 16 octets contain the CFB Initialization Vector. The rest of the message is encrypted using CFB and protected by the HMAC.

The next 16 bits are not used. They SHOULD be set to zero and MUST be ignored in received Confirm1 messages.

The next 8 bits contain the signature length. If no SAS signature (described in Section 8.3) is present, all bits are set to zero. The signature length is in words and includes the signature type block. If the calculated signature octet count is not a multiple of 4, zeros are added to pad it out to a word boundary. If no signature block is present, the overall length of the Confirm1 or Confirm2 Message will be set to 11 words.

The next 8 bits are used for flags. Undefined flags are set to zero and ignored. Three flags are currently defined. The Disclosure Flag (D) is a Boolean bit defined in Appendix B. The Allow Clear flag (A) is a Boolean bit defined in Section 5.6. The SAS Verified flag (V) is a Boolean bit defined in Section 8. The cache expiration interval is an unsigned 32 bit integer of the number of seconds that the newly generated cached shared secret, rs1, should be stored.

If the signature length (in words) is non-zero, a signature type block will be present along with a signature block. Next is the signature block.

CFB [11] mode is applied with a feedback length of 128-bits, a full cipher block, and the final block is truncated to match the exact length of the encrypted data. The CFB Initialization Vector is a 128 bit random nonce. The block cipher algorithm and the key size is the same as what was negotiated for the media encryption. CFB is used to encrypt the part of the Confirm1 message beginning after the CFB IV to the end of the message (the encrypted region is enclosed by "======" in Figure 8).

The responder uses the zrtpkeyr to encrypt the Confirm1 message. The initiator uses the zrtpkeyi to encrypt the Confirm2 message.

Figure 8. Confirm1 and Confirm2 message format

6.8. Conf2ACK message

The Conf2ACK message is sent in response to a valid Confirm2 message. The message format for the Conf2ACK is shown in Figure 9. The receipt of a Conf2ACK stops retransmission of the Confirm2 message.

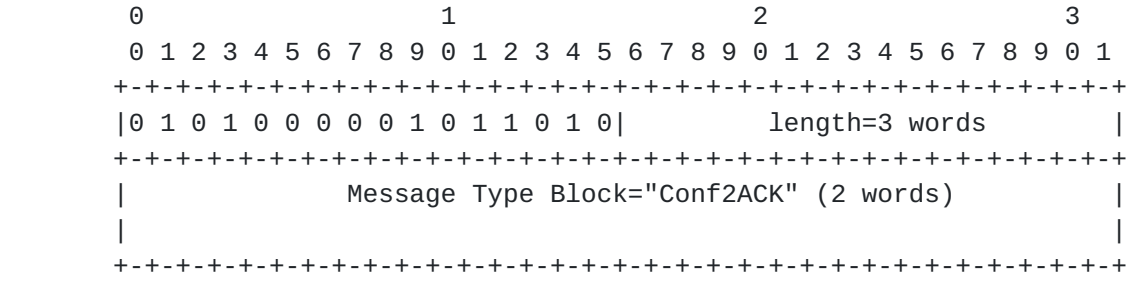

Figure 9. Conf2ACK message format

6.9. GoClear message

The GoClear message is sent to terminate an in-process ZRTP key agreement exchange or optionally to switch from SRTP to RTP. The format is shown in Figure 10 below. The Reason String is a 16 character string which contains the reason for the switch to clear. If the GoClear is sent due to a protocol error, the reason phrase is generated to describe the reason. The Reason String can be logged or rendered for human consumption. If the GoClear is sent due to a user interface selection, the reason is "User Request".

If the GoClear is sent to switch from SRTP back to RTP, the The clear_hmac is used to authenticate the GoClear message so that bogus GoClear messages introduced by an attacker can be detected and discarded.

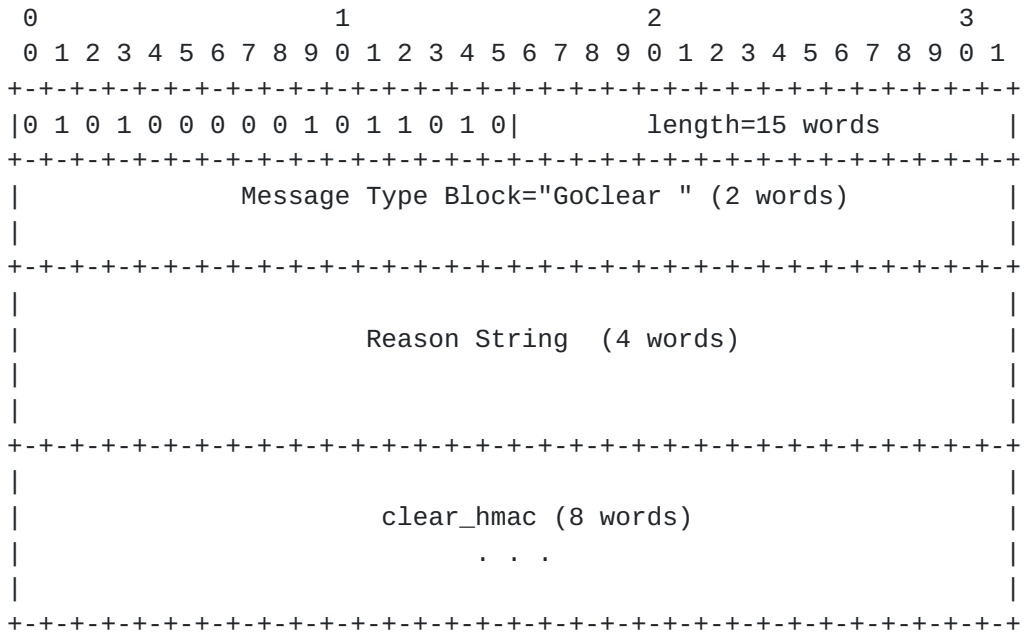

Figure 10. GoClear message format

6.10. ClearACK message

The ClearACK message is sent to acknowledge receipt of a GoClear. A ClearACK is only sent if the clear_hmac from the GoClear message is authenticated. Otherwise, no response is returned. The format is shown in Figure 11.

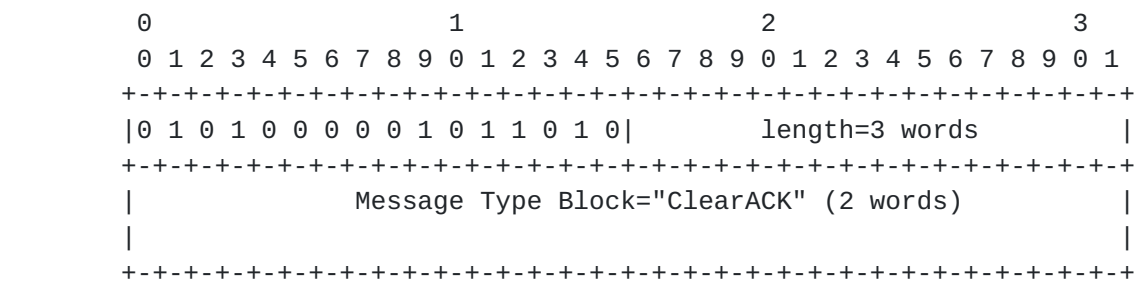

Figure 11. ClearACK message format

7. Retransmissions

ZRTP uses two retransmission timers T1 and T2. T1 is used for retransmission of Hello messages, when the support of ZRTP by the other endpoint may not be known. T2 is used in retransmissions of all the other ZRTP messages with the exception of GoClear.

All message retransmissions MUST be identical to the initial message including nonces, public values, etc; otherwise, hashes of the message sequences may not agree.

Practical experience has shown that RTP packet loss at the start of an RTP session can be extremely high. Since the entire ZRTP message exchange occurs during this period, the defined retransmission scheme is defined to be aggressive. Since ZRTP packets with the exception of the DHPart1 and DHPart2 messages are small, this should have minimal effect on overall bandwidth utilization of the media session.

Hello ZRTP requests are retransmitted at an interval that starts at T1 seconds and doubles after every retransmission, capping at 200ms. A Hello message is retransmitted 20 times before giving up. T1 has a recommended value of 50 ms. Retransmission of a Hello ends upon receipt of a HelloACK or Commit message.

Non-Hello ZRTP requests are retransmitted only by the initiator - that is, only Commit, DHPart2, and Confirm2 are retransmitted if the corresponding message from the responder, DHPart1, Confirm1, and Conf2ACK, are not received. Non-Hello ZRTP messages are retransmitted at an interval that starts at T2 seconds and doubles after every retransmission, capping at 600ms. Only the ZRTP initiator performs retransmissions. Each message is retransmitted 10 times before giving up and resuming a normal RTP session. T2 has a default value of 150ms. Each message has a response message that stops retransmissions, as shown in Table 7. The high value of T2 means that retransmissions will likely only occur with packet loss.

A GoClear message is retransmitted at 500ms intervals until a ClearACK message is received.

Message	Acknowledgement Message
Hello	HelloACK or Commit
Commit	DHPart1 or Confirm1
DHPart2	Confirm1
Confirm1	Confirm2
Confirm2	Conf2ACK
GoClear	ClearACK

Table 7. Retransmitted ZRTP Messages and Responses

8. Short Authentication String

This section will discuss the implementation of the Short Authentication String, or SAS in ZRTP. The SAS can be verified by the human users reading the string aloud, exchanging and comparing over an integrity-protected signaling channel using the a=zrtp-sas attribute, or validating a digital signature exchanged in the Confirm1 or Confirm2 messages.

The rendering of the SAS value to the user depends on the SAS Type agreed upon in the Commit message. For the SAS Type of base32, the last 20 bits of the sasvalue are rendered as a form of base32 encoding known as libbase32 [9]. The purpose of base32 is to represent arbitrary sequences of octets in a form that is as convenient as possible for human users to manipulate. As a result, the choice of characters is slightly different from base32 as defined in RFC 3548. The last 20 bits of the sasvalue results in four base32 characters which are rendered to both ZRTP endpoints. Other SAS Types may be defined to render the SAS value in other ways.

The SAS SHOULD be rendered to the user for authentication. In addition, the SAS SHOULD be sent in a subsequent offer/answer exchange (a re-INVITE in SIP) after the completion of ZRTP exchange using the ZRTP SAS SDP attributes defined in Appendix A.

The SAS is not a secret value, but it must be compared to see if it matches at both ends of the communications channel. The two users read it aloud to their partners to see if it matches. This allows detection of a man-in-the-middle (MITM) attack.

8.1. SAS Verified Flag

The SAS Verified flag (V) is set based on the user indicating that SAS comparison has been successfully performed. The SAS Verified flag is exchanged securely in the Confirm1 and Confirm2 messages of the next session. In other words, each party sends the SAS Verified flag from the previous session in the Confirm message of the current session. It is perfectly reasonable to have a ZRTP endpoint that never sets the SAS Verified flag, because it would require adding complexity to the user interface to allow the user to set it. The SAS Verified flag is not required to be set, but if it is available to the client software, it allows for the possibility that the client software could render to the user that the SAS verify procedure was carried out in a previous session.

Regardless of whether there is a user interface element to allow the user to set the SAS Verified flag, it is worth caching a shared secret, because doing so reduces opportunities for an attacker in the next call.

If at any time the users carry out the SAS comparison procedure, and it actually fails to match, then this means there is a very resourceful man in the middle. If this is the first call, the MITM was there on the first call, which is impressive enough. If it happens in a later call, it also means the MITM must also know the cached shared secret, because you could not have carried out any voice traffic at all unless the session key was correctly computed and is also known to the attacker. This implies the MITM must have been present in all the previous sessions, since the initial establishment of the first shared secret. This is indeed a resourceful attacker. It also means that if at any time he ceases his participation as a MITM on one of your calls, the protocol will detect that the cached shared secret is no longer valid -- because it was really two different shared secrets all along, one of them between Alice and the attacker, and the other between the attacker and Bob. The continuity of the cached shared secrets make it possible for us to detect the MITM when he inserts himself into the ongoing relationship, as well as when he leaves. Also, if the attacker tries to stay with a long lineage of calls, but fails to execute a DH MITM attack for even one missed call, he is permanently excluded. He can no longer resynchronize with the chain of cached shared secrets.

Some sort of user interface element (maybe a checkbox) is needed to allow the user to tell the software the SAS verify was successful, causing the software to set the SAS Verified flag (V), which (together with our cached shared secret) obviates the need to perform the SAS procedure in the next call. An additional user interface element can be provided to let the user tell the software he detected

an actual SAS mismatch, which indicates a MITM attack. The software can then take appropriate action, clearing the SAS Verified flag, and erase the cached shared secret from this session. It is up to the implementer to decide if this added user interface complexity is warranted.

If the SAS matches, it means there is no MITM, which also implies it is now safe to trust a cached shared secret for later calls. If inattentive users don't bother to check the SAS, it means we don't know whether there is or is not a MITM, so even if we do establish a new cached shared secret, there is a risk that our potential attacker may have a subsequent opportunity to continue inserting himself in the call, until we finally get around to checking the SAS. If the SAS matches, it means no attacker was present for any previous session since we started propagating cached shared secrets, because this session and all the previous sessions were also authenticated with a continuous lineage of shared secrets.

8.2. Signing the SAS

The SAS MAY be signed and the signature sent using the Confirm1 or Confirm2 messages. The signature algorithm is also sent in the Confirm1 or Confirm2 message, along with the length of the signature. The key types and signature algorithms are for future study. The signature is calculated over the 64 bit sasvalue. The signatures exchanged in the encrypted Confirm1 or Confirm2 messages MAY be used to authenticate the ZRTP exchange.

9. IANA Considerations

This specification defines two new SDP [10] attributes in Appendix A. The IANA registration of ZRTP SDP attribute:

Contact name: Phil Zimmermann <prz@mit.edu>

Attribute name: "zrtp-zid".

Type of attribute: Session level or Media level.

Subject to charset: Not.

Purpose of attribute: The 'zrtp-zid' indicates that a UA supports the ZRTP protocol and provides the ZID of the UA.

Allowed attribute values: Hex.

IANA registration of the ZRTP SAS SDP attribute:

Contact name: Phil Zimmermann <prz@mit.edu>

Attribute name: "zrtp-sas".

Type of attribute: Media level.

Subject to charset: Yes.

Purpose of attribute: The 'zrtp-sas' is used to convey the ZRTP SAS string and value. The string is identical to that rendered to the users. The value is the 64 bit SAS encoded as hex.

Allowed attribute values: String and Hex.

10. Security Considerations

This document is all about securely keying SRTP sessions. As such, security is discussed in every section.

Most secure phones rely on a Diffie-Hellman exchange to agree on a common session key. But since DH is susceptible to a man-in-the-middle (MITM) attack, it is common practice to provide a way to authenticate the DH exchange. In some military systems, this is done by depending on digital signatures backed by a centrally-managed PKI. A decade of industry experience has shown that deploying centrally managed PKIs can be a painful and often futile experience. PKIs are just too messy, and require too much activation energy to get them started. Setting up a PKI requires somebody to run it, which is not practical for an equipment provider. A service provider like a carrier might venture down this path, but even then you have to deal with cross-carrier authentication, certificate revocation lists, and other complexities. It is much simpler to avoid PKIs altogether, especially when developing secure commercial products. It is therefore more common for commercial secure phones in the PSTN world to augment the DH exchange with a Short Authentication String (SAS) combined with a hash commitment at the start of the key exchange, to shorten the length of SAS material that must be read aloud. No PKI is required for this approach to authenticating the DH exchange. The AT&T TSD 3600, Eric Blossom's COMSEC secure phones [15], PGPfone [13], and CryptoPhone [16] are all examples of products that took this simpler lightweight approach.

The main problem with this approach is inattentive users who may not execute the voice authentication procedure, or unattended secure phone calls to answering machines that cannot execute it.

Additionally, some people worry about voice spoofing. But it is a mistake to think this is simply an exercise in voice impersonation (perhaps this could be called the "Rich Little" attack). Although there are digital signal processing techniques for changing a person's voice, that does not mean a man-in-the-middle attacker can safely break into a phone conversation and inject his own short authentication string (SAS) at just the right moment. He doesn't know exactly when or in what manner the users will choose to read aloud the SAS, or in what context they will bring it up or say it, or even which of the two speakers will say it, or if indeed they both will say it. In addition, some methods of rendering the SAS involve using a list of words such as the PGP word list, in a manner analogous to how pilots use the NATO phonetic alphabet to convey information. This can make it even more complicated for the attacker, because these words can be worked into the conversation in unpredictable ways. Remember that the attacker places a very high value on not being detected, and if he makes a mistake, he doesn't get to do it over. Some people have raised the question that even if the attacker lacks voice impersonation capabilities, it may be unsafe for people who don't know each other's voices to depend on the SAS procedure. This is not as much of a problem as it seems, because it isn't necessary that they recognize each other by their voice, it's only necessary that they detect that the voice used for the SAS procedure matches the voice in the rest of the phone conversation.

A popular and field-proven approach is used by SSH (Secure Shell) [18], which Peter Gutmann likes to call the "baby duck" security model. SSH establishes a relationship by exchanging public keys in the initial session, when we assume no attacker is present, and this makes it possible to authenticate all subsequent sessions. A successful MITM attacker has to have been present in all sessions all the way back to the first one, which is assumed to be difficult for the attacker. All this is accomplished without resorting to a centrally-managed PKI.

We use an analogous baby duck security model to authenticate the DH exchange in ZRTP. We don't need to exchange persistent public keys, we can simply cache a shared secret and re-use it to authenticate a long series of DH exchanges for secure phone calls over a long period of time. If we read aloud just one SAS, and then cache a shared secret for later calls to use for authentication, no new voice authentication rituals need to be executed. We just have to remember we did one already.

If we ever lose this cached shared secret, it is no longer available for authentication of DH exchanges, so we would have to do a new SAS procedure and start over with a new cached shared secret. Then we could go back to omitting the voice authentication on later calls.

A particularly compelling reason why this approach is attractive is that SAS is easiest to implement when a GUI or some sort of display is available, which raises the question of what to do when no display is available. We envision some products that implement secure VoIP via a local network proxy, which lacks a display in many cases. If we take an approach that greatly reduces the need for a SAS in each and every call, we can operate in GUI-less products with greater ease.

It's a good idea to force your opponent to have to solve multiple problems in order to mount a successful attack. Some examples of widely differing problems we might like to present him with are: Stealing a shared secret from one of the parties, being present on the very first session and every subsequent session to carry out an active MITM attack, and solving the discrete log problem. We want to force the opponent to solve more than one of these problems to succeed.

ZRTP can use different kinds of shared secrets. Each type of shared secret is determined by a different method. All of the shared secrets are hashed together to form a session key to encrypt the call. An attacker must defeat all of the methods in order to determine the session key.

First, there is the shared secret determined entirely by a Diffie-Hellman key agreement. It changes with every call, based on random numbers. An attacker may attempt a classic DH MITM attack on this secret, but we can protect against this by displaying and reading aloud a SAS, combined with adding a hash commitment at the beginning of the DH exchange.

Second, there is an evolving shared secret, or ongoing shared secret that is automatically changed and refreshed and cached with every new session. We will call this the cached shared secret, or sometimes the retained shared secret. Each new image of this ongoing secret is a non-invertable function of its previous value and the new secret derived by the new DH agreement. It's possible that no cached shared secret is available, because there were no previous sessions to inherit this value from, or because one side loses its cache.

There are other approaches for key agreement for SRTP that compute a shared secret using information in the signaling. For example, [20] describes how to carry a MIKEY (Multimedia Internet KEYing) [21] payload in SDP [10]. Or [19] describes directly carrying SRTP keying and configuration information in SDP. ZRTP does not rely on the signaling to compute a shared secret, but If a client does produce a shared secret via the signaling, and makes it available to the ZRTP protocol, ZRTP can make use of this shared secret to augment the list

of shared secrets that will be hashed together to form a session key. This way, any security weaknesses that might compromise the shared secret contributed by the signaling will not harm the final resulting session key.

There may also be a static shared secret that the two parties agree on out-of-band in advance. A hashed passphrase would suffice.

The shared secret provided by the signaling (if available), the shared secret computed by DH, and the cached shared secret are all hashed together to compute the session key for a call. If the cached shared secret is not available, it is omitted from the hash computation. If the signaling provides no shared secret, it is also omitted from the hash computation.

No DH MITM attack can succeed if the ongoing shared secret is available to the two parties, but not to the attacker. This is because the attacker cannot compute a common session key with either party without knowing the cached secret component, even if he correctly executes a classic DH MITM attack. Mixing in the cached shared secret for the session key calculation allows it to act as an implicit authenticator to protect the DH exchange, without requiring additional explicit HMACs to be computed on the DH parameters. If the cached shared secret is available, a MITM attack would be instantly detected by the failure to achieve a shared session key, resulting in undecryptable packets. The protocol can easily detect this. It would be more accurate to say that the MITM attack is not merely detected, but thwarted.

When adding the complexity of additional shared secrets beyond the familiar DH key agreement, we must make sure the lack of availability of the cached shared secret cannot prevent a call from going through, and we must also prevent false alarms that claim an attack was detected.

An small added benefit of using these cached shared secrets to mix in with the session keys is that it augments the entropy of the session key. Even if limits on the size of the DH exchange produces a session key with less than 256 bits of real work factor, the added entropy from the cached shared secret can bring up all the subsequent session keys to the full 256-bit AES key strength, assuming no attacker was present in the first call.

We could have authenticated the DH exchange the same way SSH does it, with digital signatures, caching public keys instead of shared secrets. But this approach with caching shared secrets seemed a bit simpler, requiring less CPU time for low-powered mobile platforms because it avoids an added digital signature step.

The ZRTP SDP attributes convey information through the signaling that is already available in clear text through the media path. For example, the ZRTP flag is equivalent to sending a ZRTP Hello message. The SAS is calculated from a hash of material from ZRTP messages sent over the media path. As a result, none of the ZRTP SDP attributes require confidentiality from the signaling.

The ZRTP SAS attributes can use the signaling channel as an out-of-band authentication mechanism. This authentication is only useful if the signaling channel has end-to-end integrity protection. Note that the SIP Identity header field [23] provides middle-to-end integrity protection across SDP message bodies which provides useful protection for ZRTP SAS attributes.

11. Acknowledgments

The authors would like to thank Bryce Wilcox-O'Hearn for his contributions to the design of this protocol, and to thank Jon Peterson, Colin Plumb, Hal Finney, Colin Perkins, and Dan Wing for their helpful comments and suggestions. Also thanks to David McGrew, Roni Even, Viktor Krikun, Werner Dittmann, Allen Pulsifer, Klaus Peters, and Abhishek Arya for their feedback and comments.

12. Appendix A - Signaling Interactions

This section discusses how ZRTP, SIP, and SDP work together.

The signaling secret (sigs) can be derived from SIP signaling and passed from the signaling protocol used to establish the RTP session to ZRTP. Its the dialog identifier of a Secure SIP (sips) session: a string composed of Call-ID and the local and remote tags. It can be considered a secret because it is always transported using TLS and is randomly generated for each SIP call. The local and remote tags are sorted in ascending order in the hash. From the definitions in RFC 3261 [17]:

sigs = hash(call-id | tag1 | tag2)

Note: the dialog identifier of a non-secure SIP session should not be considered a signaling secret as it has no confidentiality protection.

Note: The signaling secret secret may not be regarded as having adequate entropy for cryptographic protection without augmentation by key material from other sources.

For the SRTP secret (srtps), it is the SRTP master key and salt. This information may have been passed in the signaling using [20] or [19], for example:

srtps = hash(SRTP master key | SRTP master salt)

Note that ZRTP may be implemented without coupling with the SIP signaling. For example, ZRTP can be implemented as a "bump in the wire" or as a "bump in the stack" in which RTP sent by the SIP UA is converted to ZRTP. In these cases, the SIP UA will have no knowledge of ZRTP. As a result, the signaling path discovery mechanisms introduced in this section should not be definitive - they are a hint. Despite the absence of an indication of ZRTP support in an offer or answer, a ZRTP endpoint SHOULD still send Hello messages.

ZRTP endpoints which have control over the signaling path include a ZRTP SDP attributes in their SDP offers and answers. The ZRTP attribute, a=zrtp-id is a flag to indicate support for ZRTP. There are a number of potential uses for this attribute. It is useful when signaling elements would like to know when ZRTP may be utilized by endpoints. It is also useful if endpoints support multiple methods of SRTP key management. The ZRTP attribute can be used to ensure that these key management approaches work together instead of against each other. For example, if only one endpoint supports ZRTP but both support another method to key SRTP, then the other method will be used instead. When used in parallel, an SRTP secret carried in an a=keymgt [20] or a=crypto [19] attribute can be used as a shared secret for the srtp_secret. The ZRTP attribute is also used to signal to an intermediary ZRTP device not to act as a ZRTP endpoint, as discussed in Appendix C.

The a=zrtp-zid attribute can be included at a media level or at the session level. It indicates support of ZRTP and provides the ZID encoded in hex of the endpoint. When used at the media level, it indicates that ZRTP is supported on this media stream. When used at the session level, it indicates that ZRTP is supported in all media streams in the session described by the offer or answer and that the same ZID will be used for both streams.

In some scenarios, it is desirable for a signaling intermediary to be able to validate the SAS on behalf of the user. This could be due to an endpoint which has a user interface unable to render the SAS. Or, this could be a protection by an organization against lazy users who never check the SAS. Using either the ZRTP SAS or ZRTP SASvalue attribute, the SAS check can be performed without requiring the human users to speak the SAS. Note that this check can only be relied on if the signaling path has end-to-end integrity protection.

The ZRTP SAS attribute a=zrtp-sas is a Media level SDP attribute that can be used to carry the SAS string and value. The string is identical to that rendered to the user while contents of the string passed depends on the negotiated SAS Type. The value is the 64 bit SAS value encoded as hex. Since the SAS is not known at the start of a session, the a=zrtp-sas attribute will never be present in the initial offer/answer exchange. After the ZRTP exchange has completed, the SAS is known and can be exchanged over the signaling using a second offer/answer exchange (a re-INVITE in SIP terms). Note that the SAS is not a secret and as such does not need confidentiality protection when sent over the signaling path.

The ABNF for the ZRTP attribute is as follows:

```
zrtp-attribute        = "a=zrtp-zid:" zid-value

zid-value             = 1*(HEXDIG)
```

The ABNF for the ZRTP SAS attribute is as follows:

```
zrtp-sas-attribute    = "a=zrtp-sas:" sas-string sas-value

sas-string            = non-ws-string

non-ws-string         = 1*(VCHAR/%x80-FF)
                         ;string of visible characters

sas-value             = 1*(HEXDIG)
```

Example of the ZRTP attribute in an initial SDP offer or answer used at the session level:

```
v=0
o=bob 2890844527 2890844527 IN IP4 client.biloxi.example.com
s=
c=IN IP4 client.biloxi.example.com
a=zrtp-zid:4cc3ffe30efd02423cb054e5
t=0 0
m=audio 3456 RTP/AVP 97 33
a=rtpmap:97 iLBC/8000
a=rtpmap:33 no-op/8000
```

Example of the ZRTP SAS and SASvalue attribute in a subsequent SDP offer or answer used at the media level. Note that the a=zrtp-id attribute doesn't provide any additional information when used with

the SAS and SASvalue attributes but does not do any harm:

```
v=0
o=bob 2890844527 2890844528 IN IP4 client.biloxi.example.com
s=
c=IN IP4 client.biloxi.example.com
a=zrtp-zid:4cc3ffe30efd02423cb054e5
t=0 0
m=audio 3456 RTP/AVP 97 33
a=rtpmap:97 iLBC/8000
a=rtpmap:33 no-op/8000
a=zrtp-sas: opzf 5e017f3a6563876a
```

Another example showing a second media stream being added to the session. A second DH exchange is performed (instead of using the Preshared mode) resulting in a second set of ZRTP SAS and SASvalue attributes.

```
v=0
o=bob 2890844527 2890844528 IN IP4 client.biloxi.example.com
s=
c=IN IP4 client.biloxi.example.com
a=zrtp-zid:4cc3ffe30efd02423cb054e5
t=0 0
m=audio 3456 RTP/AVP 97 33
a=rtpmap:97 iLBC/8000
a=rtpmap:33 no-op/8000
a=zrtp-sas: opzf 5e017f3a6563876a
m=video 51372 RTP/AVP 31 33
a=rtpmap:31 H261/90000
a=rtpmap:33 no-op/8000
a=zrtp-sas: gwif e1027fa9f865221c
```

13. Appendix B - The ZRTP Disclosure flag

There are no back doors defined in the ZRTP protocol specification. The designers of ZRTP would like to discourage back doors in ZRTP-enabled products. However, despite the lack of back doors in the actual ZRTP protocol, it must be recognized that a ZRTP implementer might still deliberately create a rogue ZRTP-enabled product that implements a back door outside the scope of the ZRTP protocol. For example, they could create a product that discloses the SRTP session key generated using ZRTP out-of-band to a third party. They may even have a legitimate business reason to do this for some customers.

For example, some environments have a need to monitor or record calls, such as stock brokerage houses who want to discourage insider

trading, or special high security environments with special needs to monitor their own phone calls. We've all experienced automated messages telling us that "This call may be monitored for quality assurance". A ZRTP endpoint in such an environment might unilaterally disclose the session key to someone monitoring the call. ZRTP-enabled products that perform such out-of-band disclosures of the session key can undermine public confidence in the ZRTP protocol, unless we do everything we can in the protocol to alert the other user that this is happening.

If one of the parties is using a product that is designed to disclose their session key, ZRTP requires them to confess this fact to the other party through a protocol message to the other party's ZRTP client, which can properly alert that user, perhaps by rendering it in a GUI. The disclosing party does this by sending a Disclosure flag (D) in Confirm1 and Confirm2 messages as described in Sections 6.7 and 6.8.

Note that the intention here is to have the Disclosure flag identify products that are designed to disclose their session keys, not to identify which particular calls are compromised on a call-by-call basis. This is an important legal distinction, because most government sanctioned wiretap regulations require a VoIP service provider to not reveal which particular calls are wiretapped. But there is nothing illegal about revealing that a product is designed to be wiretap-friendly. The ZRTP protocol mandates that such a product "out" itself.

You might be using a ZRTP-enabled product with no back doors, but if your own GUI tells you the call is (mostly) secure, except that the other party is using a product that is designed in such a way that it may have disclosed the session key for monitoring purposes, you might ask him what brand of secure telephone he is using, and make a mental note not to purchase that brand yourself. If we create a protocol environment that requires such back-doored phones to confess their nature, word will spread quickly, and the "unseen hand" of the free market will act. The free market has effectively dealt with this in the past.

Of course, a ZRTP implementer can lie about his product having a back door, but the ZRTP standard mandates that ZRTP-compliant products MUST adhere to the requirement that a back door be confessed by sending the Disclosure flag to the other party.

There will be inevitable comparisons to Steve Bellovin's 2003 April fool's joke, when he submitted RFC 3514 [22] which defined the "Evil bit" in the IPV4 header, for packets with "evil intent". But we submit that a similar idea can actually have some merit for securing

VoIP. Sure, one can always imagine that some implementer will not be fazed by the rules and will lie, but they would have lied anyway even without the Disclosure flag. There are good reasons to believe that it will improve the overall percentage of implementations that at least tell us if they put a back door in their products, and may even get some of them to decide not to put in a back door at all. From a civic hygiene perspective, we are better off with having the Disclosure flag in the protocol.

If an endpoint stores or logs SRTP keys or information that can be used to reconstruct or recover SRTP keys after they are no longer in use (i.e. the session is active), or otherwise discloses or passes SRTP keys or information that can be used to reconstruct or recover SRTP keys to another application or device, the Disclosure flag D MUST be set in the Confirm1 or Confirm2 message.

14. Appendix C - Intermediary ZRTP Devices

This section discusses the operation of a ZRTP endpoint which is actually an intermediary. For example, consider a device which proxies both signaling and media between endpoints. There are three possible ways in which such a device could support ZRTP.

An intermediary device can act transparently to the ZRTP protocol. To do this, a device MUST pass RTP header extensions and payloads (to allow the ZRTP Flag) and non-RTP protocols multiplexed on the same port as RTP (to allow ZRTP and STUN). This is the RECOMMENDED behavior for intermediaries as ZRTP and SRTP are best when done end-to-end.

An intermediary device could implement the ZRTP protocol and act as a ZRTP endpoint on behalf of non-ZRTP endpoints behind the intermediary device. The intermediary could determine on a call-by-call basis whether the endpoint behind it supports ZRTP based on the presence or absence of the ZRTP SDP attribute flag (a=zrtp-id). For non-ZRTP endpoints, the intermediary device could act as the ZRTP endpoint using its own ZID and cache. This approach MUST only be used when there is some other security method protecting the confidentiality of the media between the intermediary and the inside endpoint, such as IPSec or physical security.

The third mode, which is NOT RECOMMENDED, is for the intermediary device to attempt to back-to-back the ZRTP protocol. In this mode, the intermediary would attempt to act as a ZRTP endpoint towards both endpoints of the media session. This approach MUST NOT be used as it will always result in a detected Man-in-the-Middle attack and will generate alarms on both endpoints and likely result in the immediate

termination of the session. It cannot be stated strongly enough that there are no usable back-to-back uses for the ZRTP protocol.

In cases where centralized media mixing is taking place, the SAS will not match when compared by the humans. However, this situation is known in the SIP signaling by the presence of the isfocus feature tag [25]. As a result, when the isfocus feature tag is present, the SAS can only be verified by comparison in the signaling or by validating signatures in the Confirm. For example, consider a audio conference call with three participants Alice, Bob, and Carol hosted on a conference bridge in Dallas. There will be three ZRTP encrypted media streams between each participant and Dallas. Each will have a different SAS. Each participant will be able to validates their SAS with the conference bridge using a=zrtp-sas or Confirm messages containing signatures.

SIP feature tags can also be used to detect if a session is established with an automaton such as an IVR, voicemail system, or speech recognition system. The display of SAS strings to users should be disabled in these cases.

It is possible that an intermediary device acting as a ZRTP endpoint might still receive ZRTP Hello and other messages from the inside endpoint. This could occur if there is another inline ZRTP device which does not include the ZRTP SDP attribute flag. If this occurs, the intermediary MUST NOT pass these ZRTP messages if it is acting as the ZRTP endpoint.

15. Appendix D - RTP Header Extension Flag for ZRTP

This specification defines a new RTP header extension used only for discovery of support for ZRTP. No ZRTP data is transported in the extension. When used, the X bit is set in the RTP header to indicate the presence of the RTP header extension.

Section 5.3.1 in RFC 3550 defines the format of an RTP Header extension. The Header extension is appended to the RTP header. The first 16 bits are an identifier for the header extension, and the following 16 bits are length of the extension header in 32 bit words. The ZRTP flag RTP header extension has the value of 0x505A and a length of 0. The format of the header extension is as shown in Figure 12.

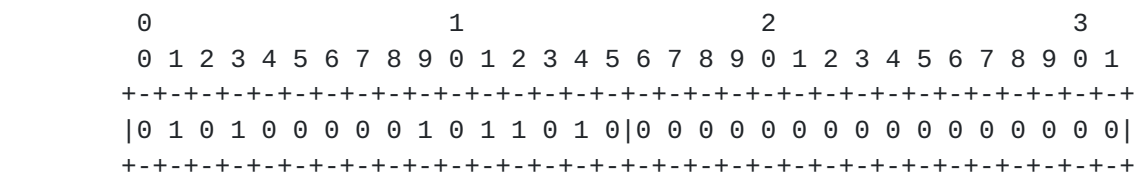

Figure 12. RTP Extension header format for ZRTP Flag

ZRTP endpoints SHOULD include the ZRTP Flag in RTP packets sent at the start of a session. For example, including the flag in the first 1 second of RTP packets sent. The inclusion of the flag MAY be ended if a ZRTP message (such as Hello) is received.

16. References

16.1. Normative References

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

[2] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003.

[3] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, March 2004.

[4] McGrew, D., "The use of AES-192 and AES-256 in Secure RTP", draft-mcgrew-srtp-big-aes-00 (work in progress), April 2006.

[5] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, May 2003.

[6] Stone, J., Stewart, R., and D. Otis, "Stream Control Transmission Protocol (SCTP) Checksum Change", RFC 3309, September 2002.

[7] Ferguson, N. and B. Schneier, "Practical Cryptography", Wiley Publishing 2003.

[8] Barker, E. and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", NIST Special Publication 800-90 DRAFT (December 2005).

[9] Wilcox, B., "Human-oriented base-32 encoding", http://cvs.sourceforge.net/viewcvs.py/libbase32/libbase32/DESIGN?rev=HEAD .

[10] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, July 2006.

[11] Dworkin, M., "Recommendation for Block Cipher: Methods and Techniques", NIST Special Publication 800-38A 2001 Edition.

16.2. Informative References

[12] Wing, D., "Media Security Requirements", draft-wing-media-security-requirements-00 (work in progress), October 2006.

[13] Zimmermann, P., "PGPfone", http://www.pgpi.org/products/pgpfone/ .

[14] Zimmermann, P., "Zfone", http://www.philzimmermann.com/zfone .

[15] Blossom, E., "The VP1 Protocol for Voice Privacy Devices Version 1.2", http://www.comsec.com/vp1-protocol.pdf .

[16] "CryptoPhone", http://www.cryptophone.de/ .

[17] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[18] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[19] Andreasen, F., Baugher, M., and D. Wing, "Session Description Protocol (SDP) Security Descriptions for Media Streams", RFC 4568, July 2006.

[20] Arkko, J., Lindholm, F., Naslund, M., Norrman, K., and E. Carrara, "Key Management Extensions for Session Description Protocol (SDP) and Real Time Streaming Protocol (RTSP)", RFC 4567, July 2006.

[21] Arkko, J., Carrara, E., Lindholm, F., Naslund, M., and K. Norrman, "MIKEY: Multimedia Internet KEYing", RFC 3830, August 2004.

[22] Bellovin, S., "The Security Flag in the IPv4 Header", RFC 3514, April 1 2003.

[23] Peterson, J. and C. Jennings, "Enhancements for Authenticated Identity Management in the Session Initiation Protocol (SIP)", RFC 4474, August 2006.

[24] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", draft-ietf-mmusic-ice-13 (work in progress), January 2007.

[25] Johnston, A. and O. Levin, "Session Initiation Protocol (SIP) Call Control - Conferencing for User Agents", BCP 119, RFC 4579, August 2006.

Authors' Addresses

Philip Zimmermann
Zfone Project

Email: prz@mit.edu

Alan Johnston (editor)
Avaya
St. Louis, MO 63124

Email: alan@sipstation.com

Jon Callas
PGP Corporation

Email: jon@pgp.com

Full Copyright Statement

Intellectual Property

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).