Stream: Internet Engineering Task Force (IETF)
RFC: 8870
Category: Standards Track
Published: October 2020
ISSN: 2070-1721
Authors:
C. Jennings *Cisco Systems*
J. Mattsson *Ericsson AB*
D. McGrew *Cisco Systems*
D. Wing *Citrix Systems, Inc.*
F. Andreasen *Cisco Systems*

RFC 8870
Encrypted Key Transport for DTLS and Secure RTP

Abstract

Encrypted Key Transport (EKT) is an extension to DTLS (Datagram Transport Layer Security) and the Secure Real-time Transport Protocol (SRTP) that provides for the secure transport of SRTP master keys, rollover counters, and other information within SRTP. This facility enables SRTP for decentralized conferences by distributing a common key to all of the conference endpoints.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Real-time Transport Protocol (RTP) is designed to allow decentralized groups with minimal control to establish sessions, such as for multimedia conferences. Unfortunately, Secure RTP (SRTP) [RFC3711] cannot be used in many minimal-control scenarios, because it requires that synchronization source (SSRC) values and other data be coordinated among all of the participants in a session. For example, if a participant joins a session that is already in progress, that participant needs to be informed of the SRTP keys along with the SSRC, rollover counter (ROC), and other details of the other SRTP sources.

The inability of SRTP to work in the absence of central control was well understood during the design of the protocol; the omission was considered less important than optimizations such as bandwidth conservation. Additionally, in many situations, SRTP is used in conjunction with a signaling system that can provide the central control needed by SRTP. However, there are several cases in which conventional signaling systems cannot easily provide all of the coordination required.

This document defines Encrypted Key Transport (EKT) for SRTP and reduces the amount of external signaling control that is needed in an SRTP session with multiple receivers. EKT securely distributes the SRTP master key and other information for each SRTP source. With this method, SRTP entities are free to choose SSRC values as they see fit and to start up new SRTP sources with new SRTP master keys within a session without coordinating with other entities via external signaling or other external means.

EKT extends DTLS and SRTP to enable a common key encryption key (called an "EKTKey") to be distributed to all endpoints, so that each endpoint can securely send its SRTP master key and current SRTP ROC to the other participants in the session. This data furnishes the information needed by the receiver to instantiate an SRTP receiver context.

EKT can be used in conferences where the central Media Distributor or conference bridge cannot decrypt the media, such as the type defined in [RFC8871]. It can also be used for large-scale conferences where the conference bridge or Media Distributor can decrypt all the media but wishes to encrypt the media it is sending just once and then send the same encrypted media to a large number of participants. This reduces the amount of CPU time needed for encryption and can be used for some optimization to media sending that use source-specific multicast.

EKT does not control the manner in which the SSRC is generated. It is only concerned with distributing the security parameters that an endpoint needs to associate with a given SSRC in order to decrypt SRTP packets from that sender.

EKT is not intended to replace external key establishment mechanisms. Instead, it is used in conjunction with those methods, and it relieves those methods of the burden of delivering the context for each SRTP source to every SRTP participant. This document defines how EKT works with the DTLS-SRTP approach to key establishment, by using keys derived from the DTLS-SRTP handshake to encipher the EKTKey in addition to the SRTP media.

2. Overview

This specification defines a way for the server in a DTLS-SRTP negotiation (see Section 5) to provide an EKTKey to the client during the DTLS handshake. The EKTKey thus obtained can be used to encrypt the SRTP master key that is used to encrypt the media sent by the endpoint. This specification also defines a way to send the encrypted SRTP master key (with the EKTKey) along with the SRTP packet (see Section 4). Endpoints that receive this and know the EKTKey can use the EKTKey to decrypt the SRTP master key, which can then be used to decrypt the SRTP packet.

One way to use this is described in the architecture defined by [RFC8871]. Each participant in the conference forms a DTLS-SRTP connection to a common Key Distributor that distributes the same EKTKey to all the endpoints. Then, each endpoint picks its own SRTP master key for the media they send. When sending media, the endpoint also includes the SRTP master key encrypted with the EKTKey in the SRTP packet. This allows all the endpoints to decrypt the media.

3. Conventions Used in This Document

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

4. Encrypted Key Transport

EKT defines a new method of providing SRTP master keys to an endpoint. In order to convey the ciphertext corresponding to the SRTP master key, and other additional information, an additional field, called the "EKTField", is added to the SRTP packets. The EKTField appears at the end of the SRTP packet. It appears after the optional authentication tag, if one is present; otherwise, the EKTField appears after the ciphertext portion of the packet.

EKT **MUST NOT** be used in conjunction with SRTP's MKI (Master Key Identifier) or with SRTP's <From, To> [RFC3711], as those SRTP features duplicate some of the functions of EKT. Senders **MUST NOT** include the MKI when using EKT. Receivers **SHOULD** simply ignore any MKI field received if EKT is in use.

This document defines the use of EKT with SRTP. Its use with the Secure Real-time Transport Control Protocol (SRTCP) would be similar, but that topic is left for a future specification. SRTP is preferred for transmitting keying material because (1) it shares fate with the transmitted media, (2) SRTP rekeying can occur without concern for RTCP transmission limits, and (3) it avoids the need for SRTCP compound packets with RTP translators and mixers.

4.1. EKTField Formats

The EKTField uses the formats defined in Figures 1 and 2 for the FullEKTField and ShortEKTField. The EKTField appended to an SRTP packet can be referred to as an "EKT tag".

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    :                                                               :
    :                        EKT Ciphertext                         :
    :                                                               :
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |   Security Parameter Index    |             Epoch             |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |            Length             |0 0 0 0 0 0 1 0|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: FullEKTField Format

```
                             0 1 2 3 4 5 6 7
                            +-+-+-+-+-+-+-+-+
                            |0 0 0 0 0 0 0 0|
                            +-+-+-+-+-+-+-+-+
```

Figure 2: ShortEKTField Format

Figure 3 shows the syntax of the EKTField, expressed in ABNF [RFC5234]. The EKTField is added to the end of an SRTP packet. The EKTPlaintext is the concatenation of SRTPMasterKeyLength, SRTPMasterKey, SSRC, and ROC, in that order. The EKTCiphertext is computed by encrypting the EKTPlaintext using the EKTKey. Future extensions to the EKTField **MUST** conform to the syntax of the ExtensionEKTField.

```
BYTE = %x00-FF

EKTMsgTypeFull = %x02
EKTMsgTypeShort = %x00
EKTMsgTypeExtension = %x03-FF ; Message type %x01 is reserved, due to
                              ; usage by legacy implementations.

EKTMsgLength = 2BYTE;

SRTPMasterKeyLength = BYTE
SRTPMasterKey = 1*242BYTE
SSRC = 4BYTE; SSRC from RTP
ROC = 4BYTE ; ROC from SRTP FOR THE GIVEN SSRC

EKTPlaintext = SRTPMasterKeyLength SRTPMasterKey SSRC ROC

EKTCiphertext = 1*251BYTE ; EKTEncrypt(EKTKey, EKTPlaintext)
Epoch = 2BYTE
SPI = 2BYTE

FullEKTField = EKTCiphertext SPI Epoch EKTMsgLength EKTMsgTypeFull

ShortEKTField = EKTMsgTypeShort

ExtensionData = 1*1024BYTE
ExtensionEKTField = ExtensionData EKTMsgLength EKTMsgTypeExtension

EKTField = FullEKTField / ShortEKTField / ExtensionEKTField
```

Figure 3: EKTField Syntax

These fields and data elements are defined as follows:

EKTPlaintext:
: This is the data that is input to the EKT encryption operation. This data never appears on the wire; it is used only in computations internal to EKT. This is the concatenation of the SRTP master key and its length, the SSRC, and the ROC.

EKTCiphertext:
: This is the data that is output from the EKT encryption operation (see Section 4.4). This field is included in SRTP packets when EKT is in use. The length of the EKTCiphertext can be larger than the length of the EKTPlaintext that was encrypted.

SRTPMasterKey:
: On the sender side, this is the SRTP master key associated with the indicated SSRC.

SRTPMasterKeyLength:
: This is the length of the SRTPMasterKey in bytes. This depends on the cipher suite negotiated for SRTP using Session Description Protocol (SDP) Offer/Answer [RFC3264] for the SRTP.

SSRC:
: On the sender side, this is the SSRC for this SRTP source. The length of this field is 32 bits. The SSRC value in the EKT tag **MUST** be the same as the one in the header of the SRTP packet to which the tag is appended.

Rollover Counter (ROC):
: On the sender side, this is set to the current value of the SRTP ROC in the SRTP context associated with the SSRC in the SRTP packet. The length of this field is 32 bits.

Security Parameter Index (SPI):
: This field indicates the appropriate EKTKey and other parameters for the receiver to use when processing the packet, within a given conference. The length of this field is 16 bits, representing a two-byte integer in network byte order. The parameters identified by this field are as follows:

  - The EKT cipher used to process the packet.
  - The EKTKey used to process the packet.
  - The SRTP master salt associated with any master key encrypted with this EKT Key. The master salt is communicated separately, via signaling, typically along with the EKTKey. (Recall that the SRTP master salt is used in the formation of Initialization Vectors (IVs) / nonces.)

Epoch:
: This field indicates how many SRTP keys have been sent for this SSRC under the current EKTKey, prior to the current key, as a two-byte integer in network byte order. It starts at zero at the beginning of a session and resets to zero whenever the EKTKey is changed (i.e., when a new SPI appears). The epoch for an SSRC increments by one every time the sender transmits a new key. The recipient of a FullEKTField **MUST** reject any future FullEKTField for this SPI and SSRC that has an epoch value equal to or lower than an epoch already seen.

Together, these data elements are called an "EKT parameter set". To avoid ambiguity, each distinct EKT parameter set that is used **MUST** be associated with a distinct SPI value.

EKTMsgLength:
: All EKT message types other than the ShortEKTField have a length as second from the last element. This is the length in octets (in network byte order) of either the FullEKTField/ExtensionEKTField including this length field and the following EKT Message Type.

Message Type:
: The last byte is used to indicate the type of the EKTField. This **MUST** be 2 for the FullEKTField format and 0 for the ShortEKTField format. If a received EKT tag has an unknown message type, then the receiver **MUST** discard the whole EKT tag.

4.2. SPIs and EKT Parameter Sets

The SPI identifies the parameters for how the EKT tag should be processed:

- The EKTKey and EKT cipher used to process the packet.
- The SRTP master salt associated with any master key encrypted with this EKT Key. The master salt is communicated separately, via signaling, typically along with the EKTKey.

Together, these data elements are called an "EKT parameter set". To avoid ambiguity, each distinct EKT parameter set that is used **MUST** be associated with a distinct SPI value. The association of a given parameter set with a given SPI value is configured by some other protocol, e.g., the DTLS-SRTP extension defined in Section 5.

4.3. Packet Processing and State Machine

At any given time, each SRTP source has associated with it a single EKT parameter set. This parameter set is used to process all outbound packets and is called the "outbound parameter set" for that SSRC. There may be other EKT parameter sets that are used by other SRTP sources in the same session, including other SRTP sources on the same endpoint (e.g., one endpoint with voice and video might have two EKT parameter sets, or there might be multiple video sources on an endpoint, each with their own EKT parameter set). All of the received EKT parameter sets **SHOULD** be stored by all of the participants in an SRTP session, for use in processing inbound SRTP traffic. If a participant deletes an EKT parameter set (e.g., because of space limitations), then it will be unable to process Full EKT Tags containing updated media keys and thus will be unable to receive media from a participant that has changed its media key.

Either the FullEKTField or ShortEKTField is appended at the tail end of all SRTP packets. The decision regarding which parameter to send and when is specified in Section 4.6.

4.3.1. Outbound Processing

See Section 4.6, which describes when to send an SRTP packet with a FullEKTField. If a FullEKTField is not being sent, then a ShortEKTField is sent so the receiver can correctly determine how to process the packet.

When an SRTP packet is sent with a FullEKTField, the EKTField for that packet is created per either the steps below or an equivalent set of steps.

1. The Security Parameter Index (SPI) field is set to the value of the SPI that is associated with the outbound parameter set.
2. The EKTPlaintext field is computed from the SRTP master key, SSRC, and ROC fields, as shown in Section 4.1. The ROC, SRTP master key, and SSRC used in EKT processing **MUST** be the same as the one used in SRTP processing.
3. The EKTCiphertext field is set to the ciphertext created by encrypting the EKTPlaintext with the EKTCipher using the EKTKey as the encryption key. The encryption process is detailed in Section 4.4.

4. Then, the FullEKTField is formed using the EKTCiphertext and the SPI associated with the EKTKey used above. Also appended are the Length and Message Type using the FullEKTField format.

   > Note: The value of the EKTCiphertext field is identical in successive packets protected by the same EKTKey and SRTP master key. This value **MAY** be cached by an SRTP sender to minimize computational effort.

The computed value of the FullEKTField is appended to the end of the SRTP packet, after the encrypted payload.

When a packet is sent with the ShortEKTField, the ShortEKTField is simply appended to the packet.

Outbound packets **SHOULD** continue to use the old SRTP master key for 250 ms after sending any new key in a FullEKTField value. This gives all the receivers in the system time to get the new key before they start receiving media encrypted with the new key. (The specific value of 250 ms is chosen to represent a reasonable upper bound on the amount of latency and jitter that is tolerable in a real-time context.)

4.3.2. Inbound Processing

When receiving a packet on an RTP stream, the following steps are applied for each received SRTP packet.

1. The final byte is checked to determine which EKT format is in use. When an SRTP packet contains a ShortEKTField, the ShortEKTField is removed from the packet and then normal SRTP processing occurs. If the packet contains a FullEKTField, then processing continues as described below. The reason for using the last byte of the packet to indicate the type is that the length of the SRTP part is not known until the decryption has occurred. At this point in the processing, there is no easy way to know where the EKTField would start. However, the whole UDP packet has been received, so instead of starting at the front of the packet, the parsing works backwards at the end of the packet, and thus the type is placed at the very end of the packet.
2. The Security Parameter Index (SPI) field is used to find the right EKT parameter set to be used for processing the packet. If there is no matching SPI, then the verification function **MUST** return an indication of authentication failure, and the steps described below are not performed. The EKT parameter set contains the EKTKey, the EKTCipher, and the SRTP master salt.
3. The EKTCiphertext is authenticated and decrypted, as described in Section 4.4, using the EKTKey and EKTCipher found in the previous step. If the EKT decryption operation returns an authentication failure, then EKT processing **MUST** be aborted. The receiver **SHOULD** discard the whole UDP packet.
4. The resulting EKTPlaintext is parsed as described in Section 4.1, to recover the SRTP master key, SSRC, and ROC fields. The SRTP master salt that is associated with the EKTKey is also retrieved. If the value of the srtp_master_salt (see Section 5.2.2) sent as part of the EKTkey is

longer than needed by SRTP, then it is truncated by taking the first N bytes from the srtp_master_salt field.

5. If the SSRC in the EKTPlaintext does not match the SSRC of the SRTP packet received, then this FullEKTField **MUST** be discarded and the subsequent steps in this list skipped. After stripping the FullEKTField, the remainder of the SRTP packet **MAY** be processed as normal.
6. The SRTP master key, ROC, and SRTP master salt from the previous steps are saved in a map indexed by the SSRC found in the EKTPlaintext and can be used for any future crypto operations on the inbound packets with that SSRC.
    - Unless the transform specifies other acceptable key lengths, the length of the SRTP master key **MUST** be the same as the master key length for the SRTP transform in use. If this is not the case, then the receiver **MUST** abort EKT processing and **SHOULD** discard the whole UDP packet.
    - If the length of the SRTP master key is less than the master key length for the SRTP transform in use and the transform specifies that this length is acceptable, then the SRTP master key value is used to replace the first bytes in the existing master key. The other bytes remain the same as in the old key. For example, the double GCM transform [RFC8723] allows replacement of the first ("end-to-end") half of the master key.
7. At this point, EKT processing has successfully completed, and the normal SRTP processing takes place.

The value of the EKTCiphertext field is identical in successive packets protected by the same EKT parameter set and the same SRTP master key, and ROC. SRTP senders and receivers **MAY** cache an EKTCiphertext value to optimize processing in cases where the master key hasn't changed. Instead of encrypting and decrypting, senders can simply copy the precomputed value and receivers can compare a received EKTCiphertext to the known value.

Section 4.3.1 recommends that SRTP senders continue using an old key for some time after sending a new key in an EKT tag. Receivers that wish to avoid packet loss due to decryption failures **MAY** perform trial decryption with both the old key and the new key, keeping the result of whichever decryption succeeds. Note that this approach is only compatible with SRTP transforms that include integrity protection.

When receiving a new EKTKey, implementations need to use the ekt_ttl field (see Section 5.2.2) to create a time after which this key cannot be used, and they also need to create a counter that keeps track of how many times the key has been used to encrypt data, to ensure that it does not exceed the T value for that cipher (see Section 4.4). If either of these limits is exceeded, the key can no longer be used for encryption. At this point, implementations need to either use call signaling to renegotiate a new session or terminate the existing session. Terminating the session is a reasonable implementation choice because these limits should not be exceeded, except under an attack or error condition.

4.4. Ciphers

EKT uses an authenticated cipher to encrypt and authenticate the EKTPlaintext. This specification defines the interface to the cipher, in order to abstract the interface away from the details of that function. This specification also defines the default cipher that is used in EKT. The default cipher described in Section 4.4.1 **MUST** be implemented, but another cipher that conforms to this interface **MAY** be used. The cipher used for a given EKTCiphertext value is negotiated using the supported_ekt_ciphers extension (see Section 5.2) and indicated with the SPI value in the FullEKTField.

An EKTCipher consists of an encryption function and a decryption function. The encryption function E(K, P) takes the following inputs:

- a secret key K with a length of L bytes, and
- a plaintext value P with a length of M bytes.

The encryption function returns a ciphertext value C whose length is N bytes, where N may be larger than M. The decryption function D(K, C) takes the following inputs:

- a secret key K with a length of L bytes, and
- a ciphertext value C with a length of N bytes.

The decryption function returns a plaintext value P that is M bytes long, or it returns an indication that the decryption operation failed because the ciphertext was invalid (i.e., it was not generated by the encryption of plaintext with the key K).

These functions have the property that D(K, E(K, P)) = P for all values of K and P. Each cipher also has a limit T on the number of times that it can be used with any fixed key value. The EKTKey **MUST NOT** be used for encryption more than T times. Note that if the same FullEKTField is retransmitted three times, that only counts as one encryption.

Security requirements for EKT ciphers are discussed in Section 6.

4.4.1. AES Key Wrap

The default EKT Cipher is the Advanced Encryption Standard (AES) Key Wrap with Padding algorithm [RFC5649]. It requires a plaintext length M that is at least one octet, and it returns a ciphertext with a length of N = M + (M mod 8) + 8 octets. It can be used with key sizes of L = 16, and L = 32 octets, and its use with those key sizes is indicated as AESKW128, or AESKW256, respectively. The key size determines the length of the AES key used by the Key Wrap algorithm. With this cipher, T=2^{48}.

Cipher	L	T
AESKW128	16	2^{48}

Cipher	L	T
AESKW256	32	2^{48}

Table 1: EKT Ciphers

As AES-128 is the mandatory-to-implement transform in SRTP, AESKW128 **MUST** be implemented for EKT. AESKW256 **MAY** be implemented.

4.4.2. Defining New EKT Ciphers

Other specifications may extend this document by defining other EKTCiphers, as described in Section 7. This section defines how those ciphers interact with this specification.

An EKTCipher determines how the EKTCiphertext field is written and how it is processed when it is read. This field is opaque to the other aspects of EKT processing. EKT ciphers are free to use this field in any way, but they **SHOULD NOT** use other EKT or SRTP fields as an input. The values of the parameters L and T **MUST** be defined by each EKTCipher. The cipher **MUST** provide integrity protection.

4.5. Synchronizing Operation

If a source has its EKTKey changed by key management, it **MUST** also change its SRTP master key, which will cause it to send out a new FullEKTField and eventually begin encrypting with it, as described in Section 4.3.1. This ensures that if key management thought the EKTKey needs changing (due to a participant leaving or joining) and communicated that to a source, the source will also change its SRTP master key, so that traffic can be decrypted only by those who know the current EKTKey.

4.6. Timing and Reliability Considerations

A system using EKT learns the SRTP master keys distributed with the FullEKTField sent with the SRTP, rather than with call signaling. A receiver can immediately decrypt an SRTP packet, provided the SRTP packet contains a FullEKTField.

This section describes how to reliably and expediently deliver new SRTP master keys to receivers.

There are three cases to consider. In the first case, a new sender joins a session and needs to communicate its SRTP master key to all the receivers. In the second case, a sender changes its SRTP master key, which needs to be communicated to all the receivers. In the third case, a new receiver joins a session already in progress and needs to know the sender's SRTP master key.

The three cases are as follows:

New sender:
: A new sender **SHOULD** send a packet containing the FullEKTField as soon as possible, always before or coincident with sending its initial SRTP packet. To accommodate packet loss, it is **RECOMMENDED** that the FullEKTField be transmitted in three consecutive packets. If the sender does not send a FullEKTField in its initial packets and receivers have not otherwise been provisioned with a decryption key, then decryption will fail and SRTP packets will be dropped until the receiver receives a FullEKTField from the sender.

Rekey:
: By sending an EKT tag over SRTP, the rekeying event shares fate with the SRTP packets protected with that new SRTP master key. To accommodate packet loss, it is **RECOMMENDED** that three consecutive packets containing the FullEKTField be transmitted.

New receiver:
: When a new receiver joins a session, it does not need to communicate its sending SRTP master key (because it is a receiver). Also, when a new receiver joins a session, the sender is generally unaware of the receiver joining the session; thus, senders **SHOULD** periodically transmit the FullEKTField. That interval depends on how frequently new receivers join the session, the acceptable delay before those receivers can start processing SRTP packets, and the acceptable overhead of sending the FullEKTField. If sending audio and video, the **RECOMMENDED** frequency is the same as the rate of intra-coded video frames. If only sending audio, the **RECOMMENDED** frequency is every 100 ms.

In general, sending EKT tags less frequently will consume less bandwidth but will increase the time it takes for a join or rekey to take effect. Applications should schedule the sending of EKT tags in a way that makes sense for their bandwidth and latency requirements.

5. Use of EKT with DTLS-SRTP

This document defines an extension to DTLS-SRTP called "SRTP EKTKey Transport", which enables secure transport of EKT keying material from the DTLS-SRTP peer in the server role to the client. This allows those peers to process EKT keying material in SRTP and retrieve the embedded SRTP keying material. This combination of protocols is valuable because it combines the advantages of DTLS, which has strong authentication of the endpoint and flexibility, along with allowing secure multi-party RTP with loose coordination and efficient communication of per-source keys.

In cases where the DTLS termination point is more trusted than the media relay, the protection that DTLS affords to EKT keying material can allow EKT keys to be tunneled through an untrusted relay such as a centralized conference bridge. For more details, see [RFC8871].

5.1. DTLS-SRTP Recap

DTLS-SRTP [RFC5764] uses an extended DTLS exchange between two peers to exchange keying material, algorithms, and parameters for SRTP. The SRTP flow operates over the same transport as the DTLS-SRTP exchange (i.e., the same 5-tuple). DTLS-SRTP combines the performance and encryption flexibility benefits of SRTP with the flexibility and convenience of DTLS-integrated key and association management. DTLS-SRTP can be viewed in two equivalent ways: as a new key management method for SRTP and as a new RTP-specific data format for DTLS.

5.2. SRTP EKT Key Transport Extensions to DTLS-SRTP

This document defines a new TLS negotiated extension called "supported_ekt_ciphers" and a new TLS handshake message type called "ekt_key". The extension negotiates the cipher to be used in encrypting and decrypting EKTCiphertext values, and the handshake message carries the corresponding key.

Figure 4 shows a message flow between a DTLS 1.3 client and server using EKT configured using the DTLS extensions described in this section. (The initial cookie exchange and other normal DTLS messages are omitted.) To be clear, EKT can be used with versions of DTLS prior to 1.3. The only difference is that in pre-1.3 TLS, stacks will not have built-in support for generating and processing ACK messages.

```
Client                                             Server

ClientHello
 + use_srtp
 + supported_ekt_ciphers
                        -------->

                                               ServerHello
                                     {EncryptedExtensions}
                                                + use_srtp
                                   + supported_ekt_ciphers
                                            {... Finished}
                        <--------

{... Finished}          -------->

                                                     [ACK]
                        <--------                 [EKTKey]

[ACK]                   -------->

|SRTP packets|          <------->           |SRTP packets|
+ <EKT tags>                                  + <EKT tags>


{} Messages protected using DTLS handshake keys

[] Messages protected using DTLS application traffic keys

<> Messages protected using the EKTKey and EKT cipher

|| Messages protected using the SRTP master key sent in
   a Full EKT Tag
```

Figure 4

In the context of a multi-party SRTP session in which each endpoint performs a DTLS handshake as a client with a central DTLS server, the extensions defined in this document allow the DTLS server to set a common EKTKey for all participants. Each endpoint can then use EKT tags encrypted with that common key to inform other endpoints of the keys it uses to protect SRTP packets. This avoids the need for many individual DTLS handshakes among the endpoints, at the cost of preventing endpoints from directly authenticating one another.

```
            Client A                 Server                 Client B

                <----DTLS Handshake---->
                <--------EKTKey---------
                                        <----DTLS Handshake---->
                                        ---------EKTKey-------->

                -------------SRTP Packet + EKT Tag------------->
                <------------SRTP Packet + EKT Tag--------------
```

5.2.1. Negotiating an EKTCipher

To indicate its support for EKT, a DTLS-SRTP client includes in its ClientHello an extension of type supported_ekt_ciphers listing the ciphers used for EKT by the client, in preference order, with the most preferred version first. If the server agrees to use EKT, then it includes a supported_ekt_ciphers extension in its ServerHello containing a cipher selected from among those advertised by the client.

The extension_data field of this extension contains an "EKTCipher" value, encoded using the syntax defined in [RFC8446]:

```
         enum {
           reserved(0),
           aeskw_128(1),
           aeskw_256(2),
         } EKTCipherType;

         struct {
             select (Handshake.msg_type) {
                 case client_hello:
                     EKTCipherType supported_ciphers<1..255>;

                 case server_hello:
                     EKTCipherType selected_cipher;
             };
         } EKTCipher;
```

5.2.2. Establishing an EKT Key

Once a client and server have concluded a handshake that negotiated an EKTCipher, the server **MUST** provide to the client a key to be used when encrypting and decrypting EKTCiphertext values. EKTKeys are sent in encrypted handshake records, using handshake type ekt_key(26). The body of the handshake message contains an EKTKey structure as follows:

```
                        struct {
                          opaque ekt_key_value<1..256>;
                          opaque srtp_master_salt<1..256>;
                          uint16 ekt_spi;
                          uint24 ekt_ttl;
                        } EKTKey;
```

The contents of the fields in this message are as follows:

ekt_key_value
: The EKTKey that the recipient should use when generating EKTCiphertext values

srtp_master_salt
: The SRTP master salt to be used with any master key encrypted with this EKT Key

ekt_spi
: The SPI value to be used to reference this EKTKey and SRTP master salt in EKT tags (along with the EKT cipher negotiated in the handshake)

ekt_ttl
: The maximum amount of time, in seconds, that this EKTKey can be used. The ekt_key_value in this message **MUST NOT** be used for encrypting or decrypting information after the TTL expires.

If the server did not provide a supported_ekt_ciphers extension in its ServerHello, then EKTKey messages **MUST NOT** be sent by the client or the server.

When an EKTKey is received and processed successfully, the recipient **MUST** respond with an ACK message as described in Section 7 of [TLS-DTLS13]. The EKTKey message and ACK **MUST** be retransmitted following the rules of the negotiated version of DTLS.

EKT **MAY** be used with versions of DTLS prior to 1.3. In such cases, to provide reliability, the ACK message is still used. Thus, DTLS implementations supporting EKT with pre-1.3 versions of DTLS will need to have explicit affordances for sending the ACK message in response to an EKTKey message and for verifying that an ACK message was received. The retransmission rules for both sides are otherwise defined by the negotiated version of DTLS.

If an EKTKey message is received that cannot be processed, then the recipient **MUST** respond with an appropriate DTLS alert.

5.3. Offer/Answer Considerations

When using EKT with DTLS-SRTP, the negotiation to use EKT is done at the DTLS handshake level and does not change the SDP Offer/Answer messaging [RFC3264].

5.4. Sending the DTLS EKTKey Reliably

The DTLS EKTKey message is sent using the retransmissions specified in Section 4.2.4 of DTLS [RFC6347]. Retransmission is finished with an ACK message, or an alert is received.

6. Security Considerations

EKT inherits the security properties of the key management protocol that is used to establish the EKTKey, e.g., the DTLS-SRTP extension defined in this document.

With EKT, each SRTP sender and receiver **MUST** generate distinct SRTP master keys. This property avoids any security concerns over the reuse of keys, by empowering the SRTP layer to create keys on demand. Note that the inputs of EKT are the same as for SRTP with key-sharing: a single key is provided to protect an entire SRTP session. However, EKT remains secure even when SSRC values collide.

SRTP master keys **MUST** be randomly generated, and [RFC4086] offers some guidance about random number generation. SRTP master keys **MUST NOT** be reused for any other purpose, and SRTP master keys **MUST NOT** be derived from other SRTP master keys.

The EKT Cipher includes its own authentication/integrity check. For an attacker to successfully forge a FullEKTField, it would need to defeat the authentication mechanisms of the EKT Cipher authentication mechanism.

The presence of the SSRC in the EKTPlaintext ensures that an attacker cannot substitute an EKTCiphertext from one SRTP stream into another SRTP stream. This mitigates the impact of cut-and-paste attacks that arise due to the lack of a cryptographic binding between the EKT tag and the rest of the SRTP packet. SRTP tags can only be cut-and-pasted within the stream of packets sent by a given RTP endpoint; an attacker cannot "cross the streams" and use an EKT tag from one SSRC to reset the key for another SSRC. The Epoch field in the FullEKTField also prevents an attacker from rolling back to a previous key.

An attacker could send packets containing a FullEKTField, in an attempt to consume additional CPU resources of the receiving system by causing the receiving system to decrypt the EKT ciphertext and detect an authentication failure. In some cases, caching the previous values of the ciphertext as described in Section 4.3.2 helps mitigate this issue.

In a similar vein, EKT has no replay protection, so an attacker could implant improper keys in receivers by capturing EKTCiphertext values encrypted with a given EKTKey and replaying them in a different context, e.g., from a different sender. When the underlying SRTP transform provides integrity protection, this attack will just result in packet loss. If it does not, then it will result in random data being fed to RTP payload processing. An attacker that is in a position to mount these attacks, however, could achieve the same effects more easily without attacking EKT.

The key encryption keys distributed with EKTKey messages are group shared symmetric keys, which means they do not provide protection within the group. Group members can impersonate each other; for example, any group member can generate an EKT tag for any SSRC. The entity that distributes EKTKeys can decrypt any keys distributed using EKT and thus any media protected with those keys.

Each EKT cipher specifies a value T that is the maximum number of times a given key can be used. An endpoint **MUST NOT** encrypt more than T different FullEKTField values using the same EKTKey. In addition, the EKTKey **MUST NOT** be used beyond the lifetime provided by the TTL described in Section 5.2.

The confidentiality, integrity, and authentication of the EKT cipher **MUST** be at least as strong as the SRTP cipher and at least as strong as the DTLS-SRTP ciphers.

Part of the EKTPlaintext is known or is easily guessable to an attacker. Thus, the EKT Cipher **MUST** resist known plaintext attacks. In practice, this requirement does not impose any restrictions on our choices, since the ciphers in use provide high security even when much plaintext is known.

An EKT cipher **MUST** resist attacks in which both ciphertexts and plaintexts can be adaptively chosen and adversaries that can query both the encryption and decryption functions adaptively.

In some systems, when a member of a conference leaves the conference, that conference is rekeyed so that the member who left the conference no longer has the key. When changing to a new EKTKey, it is possible that the attacker could block the EKTKey message getting to a particular endpoint and that endpoint would keep sending media encrypted using the old key. To mitigate that risk, the lifetime of the EKTKey **MUST** be limited by using the ekt_ttl.

7. IANA Considerations

7.1. EKT Message Types

IANA has created a new table for "EKT Message Types" in the "Real-Time Transport Protocol (RTP) Parameters" registry. The initial values in this registry are as follows:

Message Type	Value	Specification
Short	0	RFC 8870
Unassigned	1	
Full	2	RFC 8870

Message Type	Value	Specification
Unassigned	3-254	
Reserved	255	RFC 8870

Table 2: EKT Message Types

New entries in this table can be added via "Specification Required" as defined in [RFC8126]. To avoid conflicts with pre-standard versions of EKT that have been deployed, IANA **SHOULD** give preference to the allocation of even values over odd values until the even code points are consumed. Allocated values **MUST** be in the range of 0 to 254.

All new EKT messages **MUST** be defined to have a length as second from the last element, as specified.

7.2. EKT Ciphers

IANA has created a new table for "EKT Ciphers" in the "Real-Time Transport Protocol (RTP) Parameters" registry. The initial values in this registry are as follows:

Name	Value	Specification
AESKW128	0	RFC 8870
AESKW256	1	RFC 8870
Unassigned	2-254	
Reserved	255	RFC 8870

Table 3: EKT Cipher Types

New entries in this table can be added via "Specification Required" as defined in [RFC8126]. The expert **SHOULD** ensure that the specification defines the values for L and T as required in Section 4.4 of this document. Allocated values **MUST** be in the range of 0 to 254.

7.3. TLS Extensions

IANA has added supported_ekt_ciphers as a new extension name to the "TLS ExtensionType Values" table of the "Transport Layer Security (TLS) Extensions" registry:

Value: 39

Extension Name: supported_ekt_ciphers

TLS 1.3: CH, SH

Recommended: Y

Reference: RFC 8870

7.4. TLS Handshake Type

IANA has added ekt_key as a new entry in the "TLS HandshakeType" table of the "Transport Layer Security (TLS) Parameters" registry:

Value:	26
Description:	ekt_key
DTLS-OK:	Y
Reference:	RFC 8870
Comment:	

8. References

8.1. Normative References

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

[RFC3264] Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, DOI 10.17487/RFC3264, June 2002, <https://www.rfc-editor.org/info/rfc3264>.

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/info/rfc3711>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC5649] Housley, R. and M. Dworkin, "Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm", RFC 5649, DOI 10.17487/RFC5649, September 2009, <https://www.rfc-editor.org/info/rfc5649>.

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

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

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

8.2. Informative References

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

[RFC8723] Jennings, C., Jones, P., Barnes, R., and A.B. Roach, "Double Encryption Procedures for the Secure Real-Time Transport Protocol (SRTP)", RFC 8723, DOI 10.17487/RFC8723, April 2020, <https://www.rfc-editor.org/info/rfc8723>.

[RFC8871] Jones, P., Benham, D., and C. Groves, "A Solution Framework for Private Media in Privacy-Enhanced RTP Conferencing (PERC)", RFC 8871, DOI 10.17487/RFC8871, October 2020, <https://www.rfc-editor.org/info/rfc8871>.

[TLS-DTLS13] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-38, 29 May 2020, <https://tools.ietf.org/html/draft-ietf-tls-dtls13-38>.

Acknowledgments

Thank you to Russ Housley, who provided a detailed review and significant help with crafting text for this document. Thanks to David Benham, Yi Cheng, Lakshminath Dondeti, Kai Fischer, Nermeen Ismail, Paul Jones, Eddy Lem, Jonathan Lennox, Michael Peck, Rob Raymond, Sean Turner, Magnus Westerlund, and Felix Wyss for fruitful discussions, comments, and contributions to this document.

Authors' Addresses

Cullen Jennings
Cisco Systems
Email: fluffy@iii.ca

John Mattsson
Ericsson AB
Email: john.mattsson@ericsson.com

David A. McGrew
Cisco Systems
Email: mcgrew@cisco.com

Dan Wing
Citrix Systems, Inc.
Email: dwing-ietf@fuggles.com

Flemming Andreasen
Cisco Systems
Email: fandreas@cisco.com