

libSRTP 1.4 Overview and Reference Manual

David A. McGrew
mcgrew@cisco.com

Preface

The original implementation and documentation of libSRTP was written by David McGrew of Cisco Systems, Inc. in order to promote the use, understanding, and interoperability of Secure RTP. Randell Jesup contributed a working SRTCP implementation and other fixes. Alex Vanzella and Will Clark contributed changes so that the AES ICM implementation can be used for ISMA media encryption. Steve Underwood contributed x86_64 portability changes. We also give thanks to Brian Weis, Mark Baugher, Jeff Chan, Bill Simon, Douglas Smith, Bill May, Richard Prestley, Joe Tardo and others for contributions, comments, and corrections.

This reference material in this documentation was generated using the doxygen utility for automatic documentation of source code.

©2001-2005 by David A. McGrew, Cisco Systems, Inc.

Contents

1	Introduction to libSRTP	1
1.1	License and Disclaimer	1
1.2	Supported Features	2
1.3	Installing and Building libSRTP	3
1.4	Applications	4
1.5	Secure RTP Background	5
1.6	libSRTP Overview	6
1.7	Example Code	7
2	libSRTP Module Index	9
2.1	libSRTP Modules	9
3	libSRTP Data Structure Index	11
3.1	libSRTP Data Structures	11
4	libSRTP Module Documentation	13

4.1	Secure RTP	13
4.2	Secure RTCP	23
4.3	SRTP events and callbacks	25
4.4	Cryptographic Algorithms	28
4.5	Cipher Types	29
4.6	Authentication Function Types	32
4.7	Error Codes	34
4.8	Cryptographic Kernel	36
4.9	Ciphers	37
5	libSRTP Data Structure Documentation	39
5.1	crypto_policy_t Struct Reference	39
5.2	srtp_event_data_t Struct Reference	41
5.3	srtp_policy_t Struct Reference	42
5.4	ssrc_t Struct Reference	44

Chapter 1

Introduction to libSRTP

This document describes libSRTP, the Open Source Secure RTP library from Cisco Systems, Inc. RTP is the Real-time Transport Protocol, an IETF standard for the transport of real-time data such as telephony, audio, and video, defined by RFC1889. Secure RTP (SRTP) is an RTP profile for providing confidentiality to RTP data and authentication to the RTP header and payload. SRTP is an IETF Proposed Standard, and is defined in RFC 3711, and was developed in the IETF Audio/Video Transport (AVT) Working Group. This library supports all of the mandatory features of SRTP, but not all of the optional features. See the [Supported Features](#) section for more detailed information.

This document is organized as follows. The first chapter provides background material on SRTP and overview of libSRTP. The following chapters provide a detailed reference to the libSRTP API and related functions. The reference material is created automatically (using the doxygen utility) from comments embedded in some of the C header files. The documentation is organized into modules in order to improve its clarity. These modules do not directly correspond to files. An underlying cryptographic kernel provides much of the basic functionality of libSRTP, but is mostly undocumented because it does its work behind the scenes.

1.1 License and Disclaimer

libSRTP is distributed under the following license, which is included in the source code distribution. It is reproduced in the manual in case you got the library from another source.

Copyright (c) 2001-2005 Cisco Systems, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Cisco Systems, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.2 Supported Features

This library supports all of the mandatory-to-implement features of SRTP (as defined by the most recent Internet Draft). Some of these features can be selected (or de-selected) at run time by setting an appropriate policy; this is done using the structure `srtp_policy_t`. Some other behaviors of the protocol can be adapted by defining an appropriate event handler for the exceptional events; see the [SRTP events and callbacks](#) section.

Some options that are not included in the specification are supported. Most notably, the TMMH authentication function is included, though it was removed from the SRTP Internet Draft during the summer of 2002.

Some options that are described in the SRTP specification are not supported. This includes

- the Master Key Index (MKI),
- key derivation rates other than zero,
- the cipher F8,
- anti-replay lists with sizes other than 128,
- the use of the packet index to select between master keys.

The user should be aware that it is possible to misuse this library, and that the result may be that the security level it provides is inadequate. If you are implementing a feature using this library, you will want to read the Security

Considerations section of the Internet Draft. In addition, it is important that you read and understand the terms outlined in the [License and Disclaimer](#) section.

1.3 Installing and Building libSRTP

To install libSRTP, download the latest release of the distribution from `srtp.sourceforge.net`. The format of the names of the distributions are `srtp-A.B.C.tgz`, where A is the version number, B is the major release number, C is the minor release number, and `tgz` is the file extension¹ You probably want to get the most recent release. Unpack the distribution and extract the source files; the directory into which the source files will go is named `srtp`.

libSRTP uses the GNU `autoconf` and `make` utilities². In the `srtp` directory, run the configure script and then `make`:

```
./configure [ options ]  
make
```

The configure script accepts the following options:

- `-help` provides a usage summary.
- `-disable-debug` compiles libSRTP without the runtime dynamic debugging system.
- `-enable-generic-aesicm` compile in changes for ismacryp
- `-enable-syslog` use syslog for error reporting.
- `-disable-stdout` disables stdout for error reporting.
- `-enable-console` use `/dev/console` for error reporting
- `-gdoi` use GDOI key management (disabled at present).

By default, dynamic debugging is enabled and stdout is used for debugging. You can use the configure options to have the debugging output sent to syslog or the system console. Alternatively, you can define `ERR_REPORTING_FILE` in `include/conf.h` to be any other file that can be opened by libSRTP, and debug messages will be sent to it.

This package has been tested on the following platforms: Mac OS X (powerpc-apple-darwin1.4), Cygwin (i686-pc-cygwin), Solaris (sparc-sun-solaris2.6), RedHat Linux 7.1 and 9 (i686-pc-linux), and OpenBSD (sparc-unknown-openbsd2.7).

¹The extension `.tgz` is identical to `tar.gz`, and indicates a compressed tar file.

²BSD `make` will not work; if both versions of `make` are on your platform, you can invoke GNU `make` as `gmake`.

1.4 Applications

Several test drivers and a simple and portable srtp application are included in the `test/` subdirectory.

Test driver	Function tested
<code>kernel_driver</code>	crypto kernel (ciphers, auth funcs, rng)
<code>srtp_driver</code>	srtp in-memory tests (does not use the network)
<code>rdbx_driver</code>	rdbx (extended replay database)
<code>roc_driver</code>	extended sequence number functions
<code>replay_driver</code>	replay database
<code>cipher_driver</code>	ciphers
<code>auth_driver</code>	hash functions

The app `rtpw` is a simple rtp application which reads words from `/usr/dict/words` and then sends them out one at a time using `[s]rtp`. Manual srtp keying uses the `-k` option; automated key management using `gdoi` will be added later.

The usage for `rtpw` is

```
rtpw [[-d <debug>]* [-k <key> [-a][-e]] [-s | -r] dest_ip dest_port][[-l]]
```

Either the `-s` (sender) or `-r` (receiver) option must be chosen. The values `dest_ip`, `dest_port` are the IP address and UDP port to which the dictionary will be sent, respectively. The options are:

- `-s` (S)RTP sender - causes app to send words
- `-r` (S)RTP receive - causes app to receive words
- `-k <key>` use SRTP master key `<key>`, where the key is a hexadecimal value (without the leading "0x")
- `-e` encrypt/decrypt (for data confidentiality) (requires use of `-k` option as well)
- `-a` message authentication (requires use of `-k` option as well)
- `-l` list the available debug modules
- `-d <debug>` turn on debugging for module `<debug>`

In order to get a random 30-byte value for use as a key/salt pair, you can use the `rand_gen` utility in the `test/` subdirectory.

An example of an SRTP session using two `rtpw` programs follows:

```
[sh1] set k=`test/rand_gen -n 30`
[sh1] echo $k
c1eec3717da76195bb878578790af71c4ee9f859e197a414a78d5abc7451
```

```
[sh1]$ test/rtpw -s -k $k -ea 0.0.0.0 9999
Security services: confidentiality message authentication
set master key/salt to C1EEC3717DA76195BB878578790AF71C/4EE9F859E197A414A78D5ABC7451
setting SSRC to 2078917053
sending word: A
sending word: a
sending word: aa
sending word: aal
sending word: aalii
sending word: aam
sending word: Aani
sending word: aardvark
...

[sh2] set k=c1eec3717da76195bb878578790af71c4ee9f859e197a414a78d5abc7451
[sh2]$ test/rtpw -r -k $k -ea 0.0.0.0 9999
security services: confidentiality message authentication
set master key/salt to C1EEC3717DA76195BB878578790AF71C/4EE9F859E197A414A78D5ABC7451
19 octets received from SSRC 2078917053 word: A
19 octets received from SSRC 2078917053 word: a
20 octets received from SSRC 2078917053 word: aa
21 octets received from SSRC 2078917053 word: aal
...
```

1.5 Secure RTP Background

In this section we review SRTP and introduce some terms that are used in libSRTP. An RTP session is defined by a pair of destination transport addresses, that is, a network address plus a pair of UDP ports for RTP and RTCP. RTCP, the RTP control protocol, is used to coordinate between the participants in an RTP session, e.g. to provide feedback from receivers to senders. An *SRTP session* is similarly defined; it is just an RTP session for which the SRTP profile is being used. An SRTP session consists of the traffic sent to the SRTP or SRTCP destination transport addresses. Each participant in a session is identified by a synchronization source (SSRC) identifier. Some participants may not send any SRTP traffic; they are called receivers, even though they send out SRTCP traffic, such as receiver reports.

RTP allows multiple sources to send RTP and RTCP traffic during the same session. The synchronization source identifier (SSRC) is used to distinguish these sources. In libSRTP, we call the SRTP and SRTCP traffic from a particular source a *stream*. Each stream has its own SSRC, sequence number, rollover counter, and other data. A particular choice of options, cryptographic mechanisms, and keys is called a *policy*. Each stream within a session can have a distinct policy applied to it. A session policy is a collection of stream policies.

A single policy can be used for all of the streams in a given session, though the case in which a single *key* is shared

across multiple streams requires care. When key sharing is used, the SSRC values that identify the streams **must** be distinct. This requirement can be enforced by using the convention that each SRTP and SRTCP key is used for encryption by only a single sender. In other words, the key is shared only across streams that originate from a particular device (of course, other SRTP participants will need to use the key for decryption). libSRTP supports this enforcement by detecting the case in which a key is used for both inbound and outbound data.

1.6 libSRTP Overview

libSRTP provides functions for protecting RTP and RTCP. RTP packets can be encrypted and authenticated (using the [srtp_protect\(\)](#) function), turning them into SRTP packets. Similarly, SRTP packets can be decrypted and have their authentication verified (using the [srtp_unprotect\(\)](#) function), turning them into RTP packets. Similar functions apply security to RTCP packets.

The typedef `srtp_stream_t` points to a structure holding all of the state associated with an SRTP stream, including the keys and parameters for cipher and message authentication functions and the anti-replay data. A particular `srtp_stream_t` holds the information needed to protect a particular RTP and RTCP stream. This datatype is intentionally opaque in order to better separate the libSRTP API from its implementation.

Within an SRTP session, there can be multiple streams, each originating from a particular sender. Each source uses a distinct stream context to protect the RTP and RTCP stream that it is originating. The typedef `srtp_t` points to a structure holding all of the state associated with an SRTP session. There can be multiple stream contexts associated with a single `srtp_t`. A stream context cannot exist independent from an `srtp_t`, though of course an `srtp_t` can be created that contains only a single stream context. A device participating in an SRTP session must have a stream context for each source in that session, so that it can process the data that it receives from each sender.

In libSRTP, a session is created using the function [srtp_create\(\)](#). The policy to be implemented in the session is passed into this function as an [srtp_policy_t](#) structure. A single one of these structures describes the policy of a single stream. These structures can also be linked together to form an entire session policy. A linked list of [srtp_policy_t](#) structures is equivalent to a session policy. In such a policy, we refer to a single [srtp_policy_t](#) as an *element*.

An [srtp_policy_t](#) structure contains two [crypto_policy_t](#) structures that describe the cryptographic policies for RTP and RTCP, as well as the SRTP master key and the SSRC value. The SSRC describes what to protect (e.g. which stream), and the [crypto_policy_t](#) structures describe how to protect it. The key is contained in a policy element because it simplifies the interface to the library. In many cases, it is desirable to use the same cryptographic policies across all of the streams in a session, but to use a distinct key for each stream. A [crypto_policy_t](#) structure can be initialized by using either the [crypto_policy_set_rtp_default\(\)](#) or [crypto_policy_set_rtcp_default\(\)](#) functions, which set a crypto policy structure to the default policies for RTP and RTCP protection, respectively.

1.7 Example Code

This section provides a simple example of how to use libSRTP. The example code lacks error checking, but is functional. Here we assume that the value `ssrc` is already set to describe the SSRC of the stream that we are sending, and that the functions `get_rtp_packet()` and `send_srtp_packet()` are available to us. The former puts an RTP packet into the buffer and returns the number of octets written to that buffer. The latter sends the RTP packet in the buffer, given the length as its second argument.

```
srtp_t session;
srtp_policy_t policy;
octet_t key[30];

/* initialize libSRTP */
srtp_init();

/* set policy to describe a policy for an SRTP stream */
crypto_policy_set_rtp_default(&policy.rtp);
crypto_policy_set_rtcp_default(&policy.rtcp);
policy.ssrc = ssrc;
policy.key = key;
policy.next = NULL;

/* set key to random value */
crypto_get_random(key, 30);

/* allocate and initialize the SRTP session */
srtp_create(&session, policy);

/* main loop: get rtp packets, send srtp packets */
while (1) {
    char rtp_buffer[2048];
    unsigned len;

    len = get_rtp_packet(rtp_buffer);
    srtp_protect(session, rtp_buffer, &len);
    send_srtp_packet(rtp_buffer, len);
}
```


Chapter 2

libSRTP Module Index

2.1 libSRTP Modules

Here is a list of all modules:

Secure RTP	13
Secure RTCP	23
SRTP events and callbacks	25
Cryptographic Algorithms	28
Cipher Types	29
Authentication Function Types	32
Error Codes	34
Cryptographic Kernel	36
Ciphers	37

Chapter 3

libSRTP Data Structure Index

3.1 libSRTP Data Structures

Here are the data structures with brief descriptions:

crypto_policy_t (Crypto_policy_t describes a particular crypto policy that can be applied to an SRTP stream)	39
srtp_event_data_t (Srtp_event_data_t is the structure passed as a callback to the event handler function) . .	41
srtp_policy_t (Policy for an SRTP session)	42
ssrc_t (An ssrc_t represents a particular SSRC value, or a 'wildcard' SSRC)	44

Chapter 4

libSRTP Module Documentation

4.1 Secure RTP

libSRTP provides functions for protecting RTP and RTCP. See Section [libSRTP Overview](#) for an introduction to the use of the library.

Modules

- group [Secure RTCP](#)
Secure RTCP functions are used to protect RTCP traffic.
- group [SRTP events and callbacks](#)
libSRTP can use a user-provided callback function to handle events.

Data Structures

- struct [crypto_policy_t](#)
crypto_policy_t describes a particular crypto policy that can be applied to an SRTP stream.
- struct [ssrc_t](#)
An ssrc_t represents a particular SSRC value, or a 'wildcard' SSRC.

- struct `srtp_policy_t`
represents the policy for an SRTP session.

Defines

- #define `SRTP_MAX_TRAILER_LEN` `SRTP_MAX_TAG_LEN`
the maximum number of octets added by `srtp_protect()`.

Typedefs

- typedef `crypto_policy_t` `crypto_policy_t`
`crypto_policy_t` describes a particular crypto policy that can be applied to an SRTP stream.
- typedef `srtp_policy_t` `srtp_policy_t`
represents the policy for an SRTP session.
- typedef `srtp_ctx_t * srtp_t`
An `srtp_t` points to an SRTP session structure.
- typedef `srtp_stream_ctx_t * srtp_stream_t`
An `srtp_stream_t` points to an SRTP stream structure.

Enumerations

- enum `sec_serv_t` { `sec_serv_none` = 0, `sec_serv_conf` = 1, `sec_serv_auth` = 2, `sec_serv_conf_and_auth` = 3 }
`sec_serv_t` describes a set of security services.
- enum `ssrc_type_t` { `ssrc_undefined` = 0, `ssrc_specific` = 1, `ssrc_any_inbound` = 2, `ssrc_any_outbound` = 3 }
`ssrc_type_t` describes the type of an SSRC.

Functions

- `err_status_t srtp_init ()`

srtp_init() initializes the srtp library.

- `err_status_t srtp_protect (srtp_t ctx, void *rtp_hdr, int *len_ptr)`
srtp_protect() is the Secure RTP sender-side packet processing function.
- `err_status_t srtp_unprotect (srtp_t ctx, void *srtp_hdr, int *len_ptr)`
srtp_unprotect() is the Secure RTP receiver-side packet processing function.
- `err_status_t srtp_create (srtp_t *session, const srtp_policy_t *policy)`
srtp_create() allocates and initializes an SRTP session.
- `err_status_t srtp_add_stream (srtp_t session, const srtp_policy_t *policy)`
srtp_add_stream() allocates and initializes an SRTP stream within a given SRTP session.
- `err_status_t srtp_remove_stream (srtp_t session, uint32_t ssrc)`
srtp_remove_stream() deallocates an SRTP stream.
- `void crypto_policy_set_rtp_default (crypto_policy_t *p)`
crypto_policy_set_rtp_default() sets a crypto policy structure to the SRTP default policy for RTP protection.
- `void crypto_policy_set_rtcp_default (crypto_policy_t *p)`
crypto_policy_set_rtcp_default() sets a crypto policy structure to the SRTP default policy for RTCP protection.
- `err_status_t srtp_dealloc (srtp_t s)`
srtp_dealloc() deallocates storage for an SRTP session context.
- `err_status_t crypto_get_random (unsigned char *buffer, unsigned int length)`
writes a random octet string.

4.1.1 Detailed Description

4.1.2 Define Documentation

4.1.2.1 #define SRTP_MAX_TRAILER_LEN SRTP_MAX_TAG_LEN

SRTP_MAX_TRAILER_LEN is the maximum length of the SRTP trailer (authentication tag and MKI) supported by libSRTP. This value is the maximum number of octets that will be added to an RTP packet by `srtp_protect()`.

4.1.3 Typedef Documentation

4.1.3.1 typedef struct [crypto_policy_t](#) [crypto_policy_t](#)

A [crypto_policy_t](#) describes a particular cryptographic policy that can be applied to an SRTP or SRTCP stream. An SRTP session policy consists of a list of these policies, one for each SRTP stream in the session.

4.1.3.2 typedef struct [srtp_policy_t](#) [srtp_policy_t](#)

A single [srtp_policy_t](#) struct represents the policy for a single SRTP stream, and a linked list of these elements represents the policy for an entire SRTP session. Each element contains the SRTP and SRTCP crypto policies for that stream, a pointer to the SRTP master key for that stream, the SSRC describing that stream, or a flag indicating a ‘wildcard’ SSRC value, and a ‘next’ field that holds a pointer to the next element in the list of policy elements, or NULL if it is the last element.

The wildcard value `SSRC_ANY_INBOUND` matches any SSRC from an inbound stream that for which there is no explicit SSRC entry in another policy element. Similarly, the value `SSRC_ANY_OUTBOUND` will matches any SSRC from an outbound stream that does not appear in another policy element. Note that wildcard SSRCs &b cannot be used to match both inbound and outbound traffic. This restriction is intentional, and it allows libSRTP to ensure that no security lapses result from accidental re-use of SSRC values during key sharing.

Warning:

The final element of the list **must** have its ‘next’ pointer set to NULL.

4.1.3.3 typedef struct [srtp_stream_ctx_t](#)* [srtp_stream_t](#)

The typedef `srtp_stream_t` is a pointer to a structure that represents an SRTP stream. This datatype is intentionally opaque in order to separate the interface from the implementation.

An SRTP stream consists of all of the traffic sent to an SRTP session by a single participant. A session can be viewed as a set of streams.

4.1.3.4 typedef struct [srtp_ctx_t](#)* [srtp_t](#)

The typedef `srtp_t` is a pointer to a structure that represents an SRTP session. This datatype is intentionally opaque in order to separate the interface from the implementation.

An SRTP session consists of all of the traffic sent to the RTP and RTCP destination transport addresses, using the RTP/SAVP (Secure Audio/Video Profile). A session can be viewed as a set of SRTP streams, each of which originates with a different participant.

4.1.4 Enumeration Type Documentation

4.1.4.1 enum [sec_serv_t](#)

A `sec_serv_t` enumeration is used to describe the particular security services that will be applied by a particular crypto policy (or other mechanism).

Enumeration values:

- `sec_serv_none` no services
- `sec_serv_conf` confidentiality
- `sec_serv_auth` authentication
- `sec_serv_conf_and_auth` confidentiality and authentication

4.1.4.2 enum [ssrc_type_t](#)

An `ssrc_type_t` enumeration is used to indicate a type of SSRC. See [srtp_policy_t](#) for more information.

Enumeration values:

- `ssrc_undefined` Indicates an undefined SSRC type.
- `ssrc_specific` Indicates a specific SSRC value
- `ssrc_any_inbound` Indicates any inbound SSRC value (i.e. a value that is used in the function [srtp_unprotect\(\)](#))
- `ssrc_any_outbound` Indicates any outbound SSRC value (i.e. a value that is used in the function [srtp_protect\(\)](#))

4.1.5 Function Documentation

4.1.5.1 [err_status_t](#) `crypto_get_random` (`unsigned char *buffer`, `unsigned int length`)

The function call `crypto_get_random(dest, len)` writes `len` octets of random data to the location to which `dest` points, and returns an error code. This error code **must** be checked, and if a failure is reported, the data in the buffer **must not** be used.

Warning:

If the return code is not checked, then non-random data may be in the buffer. This function will fail unless it is called after `crypto_kernel_init()`.

Returns:

- `err_status_ok` if no problems occurred.
- [other] a problem occurred, and no assumptions should be made about the contents of the destination buffer.

4.1.5.2 void `crypto_policy_set_rtcp_default` (`crypto_policy_t *p`)**Parameters:**

p is a pointer to the policy structure to be set to the default policy.

The function call `crypto_policy_set_rtcp_default(&p)` sets the `crypto_policy_t` at location *p* to the SRTP default policy for RTCP protection, as defined in the specification. This function is a convenience that helps to avoid dealing directly with the policy data structure. You are encouraged to initialize policy elements with this function call. Doing so may allow your code to be forward compatible with later versions of libSRTP that include more elements in the `crypto_policy_t` datatype.

Returns:

void.

4.1.5.3 void `crypto_policy_set_rtp_default` (`crypto_policy_t *p`)**Parameters:**

p is a pointer to the policy structure to be set to the default policy.

The function call `crypto_policy_set_rtp_default(&p)` sets the `crypto_policy_t` at location *p* to the SRTP default policy for RTP protection, as defined in the specification. This function is a convenience that helps to avoid dealing directly with the policy data structure. You are encouraged to initialize policy elements with this function call. Doing so may allow your code to be forward compatible with later versions of libSRTP that include more elements in the `crypto_policy_t` datatype.

Returns:

void.

4.1.5.4 `err_status_t srtp_add_stream (srtp_t session, const srtp_policy_t * policy)`

The function call `srtp_add_stream(session, policy)` allocates and initializes a new SRTP stream within a given, previously created session, applying the policy given as the other argument to that stream.

Returns:

values:

- `err_status_ok` if stream creation succeeded.
- `err_status_alloc_fail` if stream allocation failed
- `err_status_init_fail` if stream initialization failed.

4.1.5.5 `err_status_t srtp_create (srtp_t * session, const srtp_policy_t * policy)`

The function call `srtp_create(session, policy, key)` allocates and initializes an SRTP session context, applying the given policy and key.

Parameters:

session is the SRTP session to which the policy is to be added.

policy is the `srtp_policy_t` struct that describes the policy for the session. The struct may be a single element, or it may be the head of a list, in which case each element of the list is processed. The final element of the list **must** have its 'next' field set to zero.

Returns:

- `err_status_ok` if creation succeeded.
- `err_status_alloc_fail` if allocation failed.
- `err_status_init_fail` if initialization failed.

4.1.5.6 `err_status_t srtp_dealloc (srtp_t s)`

The function call `srtp_dealloc(s)` deallocates storage for the SRTP session context *s*. This function should be called no more than one time for each of the contexts allocated by the function `srtp_create()`.

Parameters:

s is the `srtp_t` for the session to be deallocated.

Returns:

- `err_status_ok` if there no problems.
- `err_status_dealloc_fail` a memory deallocation failure occurred.

4.1.5.7 `err_status_t srtp_init ()`**Warning:**

This function **must** be called before any other srtp functions.

4.1.5.8 `err_status_t srtp_protect (srtp_t ctx, void * rtp_hdr, int * len_ptr)`

The function call `srtp_protect(ctx, rtp_hdr, len_ptr)` applies SRTP protection to the RTP packet `rtp_hdr` (which has length `*len_ptr`) using the SRTP context `ctx`. If `err_status_ok` is returned, then `rtp_hdr` points to the resulting SRTP packet and `*len_ptr` is the number of octets in that packet; otherwise, no assumptions should be made about the value of either data elements.

The sequence numbers of the RTP packets presented to this function need not be consecutive, but they **must** be out of order by less than $2^{15} = 32,768$ packets.

Warning:

This function assumes that it can write the authentication tag into the location in memory immediately following the RTP packet, and assumes that the RTP packet is aligned on a 32-bit boundary.

Parameters:

`ctx` is the SRTP context to use in processing the packet.

`rtp_hdr` is a pointer to the RTP packet (before the call); after the function returns, it points to the srtp packet.

`len_ptr` is a pointer to the length in octets of the complete RTP packet (header and body) before the function call, and of the complete SRTP packet after the call, if `err_status_ok` was returned. Otherwise, the value of the data to which it points is undefined.

Returns:

- `err_status_ok` no problems
- `err_status_replay_fail` rtp sequence number was non-increasing
- *other* failure in cryptographic mechanisms

4.1.5.9 `err_status_t srtp_remove_stream (srtp_t session, uint32_t ssrc)`

The function call `srtp_remove_stream(session, ssrc)` removes the SRTP stream with the SSRC value `ssrc` from the SRTP session context given by the argument `session`.

Parameters:

session is the SRTP session from which the stream will be removed.

ssrc is the SSRC value of the stream to be removed.

Warning:

Wildcard SSRC values cannot be removed from a session.

Returns:

- `err_status_ok` if the stream deallocation succeeded.
- [other] otherwise.

4.1.5.10 `err_status_t srtp_unprotect (srtp_t ctx, void *srtp_hdr, int *len_ptr)`

The function call `srtp_unprotect(ctx, srtp_hdr, len_ptr)` verifies the Secure RTP protection of the SRTP packet pointed to by `srtp_hdr` (which has length `*len_ptr`), using the SRTP context `ctx`. If `err_status_ok` is returned, then `srtp_hdr` points to the resulting RTP packet and `*len_ptr` is the number of octets in that packet; otherwise, no assumptions should be made about the value of either data elements.

The sequence numbers of the RTP packets presented to this function need not be consecutive, but they **must** be out of order by less than $2^{15} = 32,768$ packets.

Warning:

This function assumes that the SRTP packet is aligned on a 32-bit boundary.

Parameters:

ctx is a pointer to the `srtp_t` which applies to the particular packet.

srtp_hdr is a pointer to the header of the SRTP packet (before the call). after the function returns, it points to the rtp packet if `err_status_ok` was returned; otherwise, the value of the data to which it points is undefined.

len_ptr is a pointer to the length in octets of the complete srtp packet (header and body) before the function call, and of the complete rtp packet after the call, if `err_status_ok` was returned. Otherwise, the value of the data to which it points is undefined.

Returns:

- `err_status_ok` if the RTP packet is valid.
- `err_status_auth_fail` if the SRTP packet failed the message authentication check.
- `err_status_replay_fail` if the SRTP packet is a replay (e.g. packet has already been processed and accepted).
- [other] if there has been an error in the cryptographic mechanisms.

4.2 Secure RTCP

Secure RTCP functions are used to protect RTCP traffic.

Functions

- `err_status_t srtp_protect_rtcp` (`srtp_t` ctx, void *rtcp_hdr, int *pkt_octet_len)
srtp_protect_rtcp() is the Secure RTCP sender-side packet processing function.
- `err_status_t srtp_unprotect_rtcp` (`srtp_t` ctx, void *srtcp_hdr, int *pkt_octet_len)
srtp_unprotect_rtcp() is the Secure RTCP receiver-side packet processing function.

4.2.1 Detailed Description

RTCP is the control protocol for RTP. libSRTP protects RTCP traffic in much the same way as it does RTP traffic. The function `srtp_protect_rtcp()` applies cryptographic protections to outbound RTCP packets, and `srtp_unprotect_rtcp()` verifies the protections on inbound RTCP packets.

A note on the naming convention: `srtp_protect_rtcp()` has an `srtp_t` as its first argument, and thus has ‘`srtp_`’ as its prefix. The trailing ‘`_rtcp`’ indicates the protocol on which it acts.

4.2.2 Function Documentation

4.2.2.1 `err_status_t srtp_protect_rtcp` (`srtp_t` ctx, void * rtp_hdr, int * pkt_octet_len)

The function call `srtp_protect_rtcp(ctx, rtp_hdr, len_ptr)` applies SRTCP protection to the RTCP packet `rtp_hdr` (which has length `*len_ptr`) using the SRTCP session context `ctx`. If `err_status_ok` is returned, then `rtp_hdr` points to the resulting SRTCP packet and `*len_ptr` is the number of octets in that packet; otherwise, no assumptions should be made about the value of either data elements.

Warning:

This function assumes that it can write the authentication tag into the location in memory immediately following the RTCP packet, and assumes that the RTCP packet is aligned on a 32-bit boundary.

Parameters:

ctx is the SRTP context to use in processing the packet.

rtcp_hdr is a pointer to the RTCP packet (before the call); after the function returns, it points to the srtp packet.

pkt_octet_len is a pointer to the length in octets of the complete RTCP packet (header and body) before the function call, and of the complete SRTCP packet after the call, if `err_status_ok` was returned. Otherwise, the value of the data to which it points is undefined.

Returns:

- `err_status_ok` if there were no problems.
- [other] if there was a failure in the cryptographic mechanisms.

4.2.2.2 `err_status_t srtp_unprotect_rtcp(srtp_t ctx, void * srtcp_hdr, int * pkt_octet_len)`

The function call `srtp_unprotect_rtcp(ctx, srtcp_hdr, len_ptr)` verifies the Secure RTCP protection of the SRTCP packet pointed to by `srtcp_hdr` (which has length `*len_ptr`), using the SRTP session context `ctx`. If `err_status_ok` is returned, then `srtcp_hdr` points to the resulting RTCP packet and `*len_ptr` is the number of octets in that packet; otherwise, no assumptions should be made about the value of either data elements.

Warning:

This function assumes that the SRTCP packet is aligned on a 32-bit boundary.

Parameters:

ctx is a pointer to the `srtp_t` which applies to the particular packet.

srtcp_hdr is a pointer to the header of the SRTCP packet (before the call). After the function returns, it points to the rtp packet if `err_status_ok` was returned; otherwise, the value of the data to which it points is undefined.

pkt_octet_len is a pointer to the length in octets of the complete SRTCP packet (header and body) before the function call, and of the complete rtp packet after the call, if `err_status_ok` was returned. Otherwise, the value of the data to which it points is undefined.

Returns:

- `err_status_ok` if the RTCP packet is valid.
- `err_status_auth_fail` if the SRTCP packet failed the message authentication check.
- `err_status_replay_fail` if the SRTCP packet is a replay (e.g. has already been processed and accepted).
- [other] if there has been an error in the cryptographic mechanisms.

4.3 SRTP events and callbacks

libSRTP can use a user-provided callback function to handle events.

Data Structures

- struct [srtp_event_data_t](#)

srtp_event_data_t is the structure passed as a callback to the event handler function

Typedefs

- typedef [srtp_event_data_t srtp_event_data_t](#)

srtp_event_data_t is the structure passed as a callback to the event handler function

- typedef void([srtp_event_handler_func_t](#))(srtp_event_data_t *data)

srtp_event_handler_func_t is the function prototype for the event handler.

Enumerations

- enum [srtp_event_t](#) { [event_src_collision](#), [event_key_soft_limit](#), [event_key_hard_limit](#), [event_packet_index_limit](#) }

srtp_event_t defines events that need to be handled

Functions

- [err_status_t srtp_install_event_handler](#) ([srtp_event_handler_func_t](#) func)

sets the event handler to the function supplied by the caller.

4.3.1 Detailed Description

libSRTP allows a user to provide a callback function to handle events that need to be dealt with outside of the data plane (see the enum `srtp_event_t` for a description of these events). Dealing with these events is not a strict necessity; they are not security-critical, but the application may suffer if they are not handled. The function `srtp_set_event_handler()` is used to provide the callback function.

A default event handler that merely reports on the events as they happen is included. It is also possible to set the event handler function to `NULL`, in which case all events will just be silently ignored.

4.3.2 Typedef Documentation

4.3.2.1 typedef struct `srtp_event_data_t srtp_event_data_t`

The struct `srtp_event_data_t` holds the data passed to the event handler function.

4.3.2.2 typedef void(`srtp_event_handler_func_t`)(`srtp_event_data_t *data`)

The typedef `srtp_event_handler_func_t` is the prototype for the event handler function. It has as its only argument an `srtp_event_data_t` which describes the event that needs to be handled. There can only be a single, global handler for all events in libSRTP.

4.3.3 Enumeration Type Documentation

4.3.3.1 enum `srtp_event_t`

The enum `srtp_event_t` defines events that need to be handled outside the ‘data plane’, such as SSRC collisions and key expirations.

When a key expires or the maximum number of packets has been reached, an SRTP stream will enter an ‘expired’ state in which no more packets can be protected or unprotected. When this happens, it is likely that you will want to either deallocate the stream (using `srtp_stream_dealloc()`), and possibly allocate a new one.

When an SRTP stream expires, the other streams in the same session are unaffected, unless key sharing is used by that stream. In the latter case, all of the streams in the session will expire.

Enumeration values:

- event_ssrc_collision* An SSRC collision occurred.
- event_key_soft_limit* An SRTP stream reached the soft key usage limit and will expire soon.
- event_key_hard_limit* An SRTP stream reached the hard key usage limit and has expired.
- event_packet_index_limit* An SRTP stream reached the hard packet limit (2^{48} packets).

4.3.4 Function Documentation

4.3.4.1 `err_status_t srtp_install_event_handler (srtp_event_handler_func_t func)`

The function call `srtp_install_event_handler(func)` sets the event handler function to the value `func`. The value `NULL` is acceptable as an argument; in this case, events will be ignored rather than handled.

Parameters:

- func* is a pointer to a function that takes an `srtp_event_data_t` pointer as an argument and returns void. This function will be used by libSRTP to handle events.

4.4 Cryptographic Algorithms

Modules

- group [Cipher Types](#)

Each cipher type is identified by an unsigned integer. The cipher types available in this edition of libSRTP are given by the #defines below.

- group [Authentication Function Types](#)

Each authentication function type is identified by an unsigned integer. The authentication function types available in this edition of libSRTP are given by the #defines below.

4.4.1 Detailed Description

This library provides several different cryptographic algorithms, each of which can be selected by using the `cipher_type_id_t` and `auth_type_id_t`. These algorithms are documented below.

Authentication functions that use the Universal Security Transform (UST) must be used in conjunction with a cipher other than the null cipher. These functions require a per-message pseudorandom input that is generated by the cipher.

The identifiers `STRONGHOLD_AUTH` and `STRONGHOLD_CIPHER` identify the strongest available authentication function and cipher, respectively. They are resolved at compile time to the strongest available algorithm. The stronghold algorithms can serve as did the keep of a medieval fortification; they provide the strongest defense (or the last refuge).

4.5 Cipher Types

Each cipher type is identified by an unsigned integer. The cipher types available in this edition of libSRTP are given by the #defines below.

Defines

- #define `NULL_CIPHER` 0
The null cipher performs no encryption.
- #define `AES_128_ICM` 1
AES-128 Integer Counter Mode (AES ICM).
- #define `SEAL` 2
SEAL 3.0.
- #define `AES_128_CBC` 3
AES-128 Integer Counter Mode (AES ICM).
- #define `STRONGHOLD_CIPHER` `AES_128_ICM`
Strongest available cipher.

Typedefs

- typedef uint32_t `cipher_type_id_t`
A `cipher_type_id_t` is an identifier for a particular cipher type.

4.5.1 Detailed Description

A `cipher_type_id_t` is an identifier for a `cipher_type`; only values given by the #defines above (or those present in the file `crypto_types.h`) should be used.

The identifier `STRONGHOLD_CIPHER` indicates the strongest available cipher, allowing an application to choose the strongest available algorithm without any advance knowledge about the available algorithms.

4.5.2 Define Documentation

4.5.2.1 `#define AES_128_CBC 3`

AES-128 ICM is the variant of counter mode that is used by Secure RTP. This cipher uses a 16-octet key and a 30-octet offset (or salt) value.

4.5.2.2 `#define AES_128_ICM 1`

AES-128 ICM is the variant of counter mode that is used by Secure RTP. This cipher uses a 16-octet key and a 30-octet offset (or salt) value.

4.5.2.3 `#define NULL_CIPHER 0`

The `NULL_CIPHER` leaves its inputs unaltered, during both the encryption and decryption operations. This cipher can be chosen to indicate that no encryption is to be performed.

4.5.2.4 `#define SEAL 2`

SEAL is the Software-Optimized Encryption Algorithm of Coppersmith and Rogaway. Nota bene: this cipher is IBM proprietary.

4.5.2.5 `#define STRONGHOLD_CIPHER AES_128_ICM`

This identifier resolves to the strongest cipher type available.

4.5.3 Typedef Documentation

4.5.3.1 `typedef uint32_t cipher_type_id_t`

A `cipher_type_id_t` is an integer that represents a particular cipher type, e.g. the Advanced Encryption Standard (AES). A `NULL_CIPHER` is available; this cipher leaves the data unchanged, and can be selected to indicate that no

encryption is to take place.

4.6 Authentication Function Types

Each authentication function type is identified by an unsigned integer. The authentication function types available in this edition of libSRTP are given by the #defines below.

Defines

- #define `NULL_AUTH` 0
The null authentication function performs no authentication.
- #define `UST_TMMHv2` 1
UST with TMMH Version 2.
- #define `UST_AES_128_XMAC` 2
(UST) AES-128 XORMAC
- #define `HMAC_SHA1` 3
HMAC-SHA1.
- #define `STRONGHOLD_AUTH` HMAC_SHA1
Strongest available authentication function.

Typedefs

- typedef uint32_t `auth_type_id_t`
An `auth_type_id_t` is an identifier for a particular authentication function.

4.6.1 Detailed Description

An `auth_type_id_t` is an identifier for an authentication function type; only values given by the #defines above (or those present in the file `crypto_types.h`) should be used.

The identifier `STRONGHOLD_AUTH` indicates the strongest available authentication function, allowing an application to choose the strongest available algorithm without any advance knowledge about the available algorithms. The stronghold algorithms can serve as did the keep of a medieval fortification; they provide the strongest defense (or the last refuge).

4.6.2 Define Documentation

4.6.2.1 `#define HMAC_SHA1 3`

HMAC_SHA1 implements the Hash-based MAC using the NIST Secure Hash Algorithm version 1 (SHA1).

4.6.2.2 `#define NULL_AUTH 0`

The NULL_AUTH function does nothing, and can be selected to indicate that authentication should not be performed.

4.6.2.3 `#define STRONGHOLD_AUTH HMAC_SHA1`

This identifier resolves to the strongest available authentication function.

4.6.2.4 `#define UST_AES_128_XMAC 2`

UST_AES_128_XMAC implements AES-128 XORMAC, using UST. Nota bene: the XORMAC algorithm is IBM proprietary.

4.6.2.5 `#define UST_TMMHv2 1`

UST_TMMHv2 implements the Truncated Multi-Modular Hash using UST. This function must be used in conjunction with a cipher other than the null cipher. with a cipher.

4.6.3 Typedef Documentation

4.6.3.1 `typedef uint32_t auth_type_id_t`

An `auth_type_id_t` is an integer that represents a particular authentication function type, e.g. HMAC-SHA1. A NULL_AUTH is available; this authentication function performs no computation, and can be selected to indicate that no authentication is to take place.

4.7 Error Codes

Enumerations

- enum `err_status_t` {
 `err_status_ok` = 0, `err_status_fail` = 1, `err_status_bad_param` = 2, `err_status_alloc_fail` = 3,
 `err_status_dealloc_fail` = 4, `err_status_init_fail` = 5, `err_status_terminus` = 6, `err_status_auth_fail` = 7,
 `err_status_cipher_fail` = 8, `err_status_replay_fail` = 9, `err_status_replay_old` = 10, `err_status_algo_fail` = 11,
 `err_status_no_such_op` = 12, `err_status_no_ctx` = 13, `err_status_cant_check` = 14, `err_status_key_expired` = 15
}

4.7.1 Detailed Description

Error status codes are represented by the enumeration `err_status_t`.

4.7.2 Enumeration Type Documentation

4.7.2.1 enum `err_status_t`

Enumeration values:

- err_status_ok* nothing to report
- err_status_fail* unspecified failure
- err_status_bad_param* unsupported parameter
- err_status_alloc_fail* couldn't allocate memory
- err_status_dealloc_fail* couldn't deallocate properly
- err_status_init_fail* couldn't initialize
- err_status_terminus* can't process as much data as requested
- err_status_auth_fail* authentication failure
- err_status_cipher_fail* cipher failure
- err_status_replay_fail* replay check failed (bad index)
- err_status_replay_old* replay check failed (index too old)
- err_status_algo_fail* algorithm failed test routine
- err_status_no_such_op* unsupported operation
- err_status_no_ctx* no appropriate context found

err_status_cant_check unable to perform desired validation

err_status_key_expired can't use key any more

4.8 Cryptographic Kernel

Modules

- group [Ciphers](#)

A generic cipher type enables cipher agility, that is, the ability to write code that runs with multiple cipher types. Ciphers can be used through the crypto kernel, or can be accessed directly, if need be.

4.8.1 Detailed Description

All of the cryptographic functions are contained in a kernel.

4.9 Ciphers

A generic cipher type enables cipher agility, that is, the ability to write code that runs with multiple cipher types. Ciphers can be used through the crypto kernel, or can be accessed directly, if need be.

Functions

- [err_status_t cipher_type_alloc](#) (cipher_type_t *ctype, cipher_t **cipher, unsigned key_len)
Allocates a cipher of a particular type.
- [err_status_t cipher_init](#) (cipher_t *cipher, const octet_t *key)
Initialized a cipher to use a particular key. May be invoked more than once on the same cipher.
- [err_status_t cipher_set_iv](#) (cipher_t *cipher, void *iv)
Sets the initialization vector of a given cipher.
- [err_status_t cipher_encrypt](#) (cipher_t *cipher, void *buf, unsigned int *len)
Encrypts a buffer with a given cipher.
- [err_status_t cipher_output](#) (cipher_t *c, octet_t *buffer, int num_octets_to_output)
Sets a buffer to the keystream generated by the cipher.
- [err_status_t cipher_dealloc](#) (cipher_t *cipher)
Deallocates a cipher.

4.9.1 Detailed Description

4.9.2 Function Documentation

4.9.2.1 [err_status_t cipher_dealloc](#) (cipher_t * cipher)

Warning:

May be implemented as a macro.

4.9.2.2 `err_status_t cipher_encrypt (cipher_t * cipher, void * buf, unsigned int * len)`**Warning:**

May be implemented as a macro.

4.9.2.3 `err_status_t cipher_init (cipher_t * cipher, const octet_t * key)`**Warning:**

May be implemented as a macro.

4.9.2.4 `err_status_t cipher_output (cipher_t * c, octet_t * buffer, int num_octets_to_output)`**Warning:**

May be implemented as a macro.

4.9.2.5 `err_status_t cipher_set_iv (cipher_t * cipher, void * iv)`**Warning:**

May be implemented as a macro.

4.9.2.6 `err_status_t cipher_type_alloc (cipher_type_t * ctype, cipher_t ** cipher, unsigned key_len)`**Warning:**

May be implemented as a macro.

Chapter 5

libSRTP Data Structure Documentation

5.1 `crypto_policy_t` Struct Reference

`crypto_policy_t` describes a particular crypto policy that can be applied to an SRTP stream.

Data Fields

- [cipher_type_id_t](#) `cipher_type`
- `int` `cipher_key_len`
- [auth_type_id_t](#) `auth_type`
- `int` `auth_key_len`
- `int` `auth_tag_len`
- [sec_serv_t](#) `sec_serv`

5.1.1 Detailed Description

A `crypto_policy_t` describes a particular cryptographic policy that can be applied to an SRTP or SRTCP stream. An SRTP session policy consists of a list of these policies, one for each SRTP stream in the session.

5.1.2 Field Documentation

5.1.2.1 `int crypto_policy_t::auth_key_len`

The length of the authentication function key in octets.

5.1.2.2 `int crypto_policy_t::auth_tag_len`

The length of the authentication tag in octets.

5.1.2.3 `auth_type_id_t crypto_policy_t::auth_type`

An integer representing the authentication function.

5.1.2.4 `int crypto_policy_t::cipher_key_len`

The length of the cipher key in octets.

5.1.2.5 `cipher_type_id_t crypto_policy_t::cipher_type`

An integer representing the type of cipher.

5.1.2.6 `sec_serv_t crypto_policy_t::sec_serv`

The flag indicating the security services to be applied.

The documentation for this struct was generated from the following file:

- `srtp.h`

5.2 srtp_event_data_t Struct Reference

srtp_event_data_t is the structure passed as a callback to the event handler function

Data Fields

- [srtp_t session](#)
- [srtp_stream_t stream](#)
- [srtp_event_t event](#)

5.2.1 Detailed Description

The struct srtp_event_data_t holds the data passed to the event handler function.

5.2.2 Field Documentation

5.2.2.1 [srtp_event_t srtp_event_data_t::event](#)

An enum indicating the type of event.

5.2.2.2 [srtp_t srtp_event_data_t::session](#)

The session in which the event happend.

5.2.2.3 [srtp_stream_t srtp_event_data_t::stream](#)

The stream in which the event happend.

The documentation for this struct was generated from the following file:

- [srtp.h](#)

5.3 srtp_policy_t Struct Reference

represents the policy for an SRTP session.

Data Fields

- [ssrc_t ssrc](#)
- [crypto_policy_t rtp](#)
- [crypto_policy_t rtcp](#)
- [octet_t * key](#)
- [srtp_policy_t * next](#)

5.3.1 Detailed Description

A single `srtp_policy_t` struct represents the policy for a single SRTP stream, and a linked list of these elements represents the policy for an entire SRTP session. Each element contains the SRTP and SRTCP crypto policies for that stream, a pointer to the SRTP master key for that stream, the SSRC describing that stream, or a flag indicating a ‘wild-card’ SSRC value, and a ‘next’ field that holds a pointer to the next element in the list of policy elements, or NULL if it is the last element.

The wildcard value `SSRC_ANY_INBOUND` matches any SSRC from an inbound stream that for which there is no explicit SSRC entry in another policy element. Similarly, the value `SSRC_ANY_OUTBOUND` will matches any SSRC from an outbound stream that does not appear in another policy element. Note that wildcard SSRCs cannot be used to match both inbound and outbound traffic. This restriction is intentional, and it allows libSRTP to ensure that no security lapses result from accidental re-use of SSRC values during key sharing.

Warning:

The final element of the list **must** have its ‘next’ pointer set to NULL.

5.3.2 Field Documentation

5.3.2.1 `octet_t* srtp_policy_t::key`

Pointer to the SRTP master key for this stream.

5.3.2.2 struct srtp_policy_t* srtp_policy_t::next

Pointer to next stream policy.

5.3.2.3 crypto_policy_t srtp_policy_t::rtcp

SRTCP crypto policy.

5.3.2.4 crypto_policy_t srtp_policy_t::rtp

SRTP crypto policy.

5.3.2.5 ssrc_t srtp_policy_t::ssrc

The SSRC value of stream, or the flags SSRC_ANY_INBOUND or SSRC_ANY_OUTBOUND if key sharing is used for this policy element.

The documentation for this struct was generated from the following file:

- srtp.h

5.4 ssrc_t Struct Reference

An ssrc_t represents a particular SSRC value, or a ‘wildcard’ SSRC.

Data Fields

- [ssrc_type_t](#) type
- [uint32_t](#) value

5.4.1 Detailed Description

An ssrc_t represents a particular SSRC value (if its type is ssrc_specific), or a wildcard SSRC value that will match all outbound SSRCs (if its type is ssrc_any_outbound) or all inbound SSRCs (if its type is ssrc_any_inbound).

5.4.2 Field Documentation

5.4.2.1 [ssrc_type_t](#) ssrc_t::type

The type of this particular SSRC

5.4.2.2 [uint32_t](#) ssrc_t::value

The value of this SSRC, if it is not a wildcard

The documentation for this struct was generated from the following file:

- srtp.h

Index

- AES_128_CBC
 - Ciphers, 30
- AES_128_ICM
 - Ciphers, 30
- auth_key_len
 - crypto_policy_t, 40
- auth_tag_len
 - crypto_policy_t, 40
- auth_type
 - crypto_policy_t, 40
- auth_type_id_t
 - Authentication, 33
- Authentication
 - auth_type_id_t, 33
 - HMAC_SHA1, 33
 - NULL_AUTH, 33
 - STRONGHOLD_AUTH, 33
 - UST_AES_128_XMAC, 33
 - UST_TMMHv2, 33
- Authentication Function Types, 32
- Cipher Types, 29
- cipher_dealloc
 - CipherImplementations, 37
- cipher_encrypt
 - CipherImplementations, 37
- cipher_init
 - CipherImplementations, 38
- cipher_key_len
 - crypto_policy_t, 40
- cipher_output
 - CipherImplementations, 38
- cipher_set_iv
 - CipherImplementations, 38
- cipher_type
 - crypto_policy_t, 40
- cipher_type_alloc
 - CipherImplementations, 38
- cipher_type_id_t
 - Ciphers, 30
- CipherImplementations
 - cipher_dealloc, 37
 - cipher_encrypt, 37
 - cipher_init, 38
 - cipher_output, 38
 - cipher_set_iv, 38
 - cipher_type_alloc, 38
- Ciphers, 37
 - AES_128_CBC, 30
 - AES_128_ICM, 30
 - cipher_type_id_t, 30
 - NULL_CIPHER, 30
 - SEAL, 30
 - STRONGHOLD_CIPHER, 30
- crypto_get_random
 - SRTP, 17
- crypto_policy_set_rtcp_default
 - SRTP, 18
- crypto_policy_set_rtp_default
 - SRTP, 18
- crypto_policy_t, 39
 - auth_key_len, 40
 - auth_tag_len, 40
 - auth_type, 40
 - cipher_key_len, 40
 - cipher_type, 40
 - sec_serv, 40
 - SRTP, 16
- Cryptographic Algorithms, 28
- Cryptographic Kernel, 36
- err_status_algo_fail
 - Error, 34
- err_status_alloc_fail

- Error, [34](#)
- err_status_auth_fail
 - Error, [34](#)
- err_status_bad_param
 - Error, [34](#)
- err_status_cant_check
 - Error, [34](#)
- err_status_cipher_fail
 - Error, [34](#)
- err_status_dealloc_fail
 - Error, [34](#)
- err_status_fail
 - Error, [34](#)
- err_status_init_fail
 - Error, [34](#)
- err_status_key_expired
 - Error, [35](#)
- err_status_no_ctx
 - Error, [34](#)
- err_status_no_such_op
 - Error, [34](#)
- err_status_ok
 - Error, [34](#)
- err_status_replay_fail
 - Error, [34](#)
- err_status_replay_old
 - Error, [34](#)
- err_status_t
 - Error, [34](#)
- err_status_terminus
 - Error, [34](#)
- Error
 - err_status_algo_fail, [34](#)
 - err_status_alloc_fail, [34](#)
 - err_status_auth_fail, [34](#)
 - err_status_bad_param, [34](#)
 - err_status_cant_check, [34](#)
 - err_status_cipher_fail, [34](#)
 - err_status_dealloc_fail, [34](#)
 - err_status_fail, [34](#)
 - err_status_init_fail, [34](#)
 - err_status_key_expired, [35](#)
 - err_status_no_ctx, [34](#)
 - err_status_no_such_op, [34](#)
 - err_status_ok, [34](#)
 - err_status_replay_fail, [34](#)
 - err_status_replay_old, [34](#)
 - err_status_t, [34](#)
 - err_status_terminus, [34](#)
- err_status_replay_old, [34](#)
- err_status_t, [34](#)
- err_status_terminus, [34](#)
- Error Codes, [34](#)
- event
 - srtp_event_data_t, [41](#)
- event_key_hard_limit
 - SRTPEvents, [27](#)
- event_key_soft_limit
 - SRTPEvents, [27](#)
- event_packet_index_limit
 - SRTPEvents, [27](#)
- event_ssrc_collision
 - SRTPEvents, [27](#)
- HMAC_SHA1
 - Authentication, [33](#)
- key
 - srtp_policy_t, [42](#)
- next
 - srtp_policy_t, [42](#)
- NULL_AUTH
 - Authentication, [33](#)
- NULL_CIPHER
 - Ciphers, [30](#)
- rtcp
 - srtp_policy_t, [43](#)
- rtp
 - srtp_policy_t, [43](#)
- SEAL
 - Ciphers, [30](#)
- sec_serv
 - crypto_policy_t, [40](#)
- sec_serv_auth
 - SRTPEvents, [17](#)
- sec_serv_conf
 - SRTPEvents, [17](#)
- sec_serv_conf_and_auth
 - SRTPEvents, [17](#)
- sec_serv_none
 - SRTPEvents, [17](#)
- sec_serv_t
 - SRTPEvents, [17](#)

- Secure RTCP, [23](#)
- Secure RTP, [13](#)
- session
 - [srtp_event_data_t](#), [41](#)
- SRTCP
 - [srtp_protect_rtcp](#), [23](#)
 - [srtp_unprotect_rtcp](#), [24](#)
- SRTP
 - [crypto_get_random](#), [17](#)
 - [crypto_policy_set_rtcp_default](#), [18](#)
 - [crypto_policy_set_rtp_default](#), [18](#)
 - [crypto_policy_t](#), [16](#)
 - [sec_serv_auth](#), [17](#)
 - [sec_serv_conf](#), [17](#)
 - [sec_serv_conf_and_auth](#), [17](#)
 - [sec_serv_none](#), [17](#)
 - [sec_serv_t](#), [17](#)
 - [srtp_add_stream](#), [18](#)
 - [srtp_create](#), [19](#)
 - [srtp_dealloc](#), [19](#)
 - [srtp_init](#), [20](#)
 - [SRTP_MAX_TRAILER_LEN](#), [15](#)
 - [srtp_policy_t](#), [16](#)
 - [srtp_protect](#), [20](#)
 - [srtp_remove_stream](#), [20](#)
 - [srtp_stream_t](#), [16](#)
 - [srtp_t](#), [16](#)
 - [srtp_unprotect](#), [21](#)
 - [ssrc_any_inbound](#), [17](#)
 - [ssrc_any_outbound](#), [17](#)
 - [ssrc_specific](#), [17](#)
 - [ssrc_type_t](#), [17](#)
 - [ssrc_undefined](#), [17](#)
- SRTP events and callbacks, [25](#)
- [srtp_add_stream](#)
 - SRTP, [18](#)
- [srtp_create](#)
 - SRTP, [19](#)
- [srtp_dealloc](#)
 - SRTP, [19](#)
- [srtp_event_data_t](#), [41](#)
 - event, [41](#)
 - session, [41](#)
 - SRTPevents, [26](#)
 - stream, [41](#)
- [srtp_event_handler_func_t](#)
 - SRTPevents, [26](#)
- [srtp_event_t](#)
 - SRTPevents, [26](#)
- [srtp_init](#)
 - SRTP, [20](#)
- [srtp_install_event_handler](#)
 - SRTPevents, [27](#)
- [SRTP_MAX_TRAILER_LEN](#)
 - SRTP, [15](#)
- [srtp_policy_t](#), [42](#)
 - key, [42](#)
 - next, [42](#)
 - rtcp, [43](#)
 - rtp, [43](#)
 - SRTP, [16](#)
 - ssrc, [43](#)
- [srtp_protect](#)
 - SRTP, [20](#)
- [srtp_protect_rtcp](#)
 - SRTCP, [23](#)
- [srtp_remove_stream](#)
 - SRTP, [20](#)
- [srtp_stream_t](#)
 - SRTP, [16](#)
- [srtp_t](#)
 - SRTP, [16](#)
- [srtp_unprotect](#)
 - SRTP, [21](#)
- [srtp_unprotect_rtcp](#)
 - SRTCP, [24](#)
- SRTPevents
 - [event_key_hard_limit](#), [27](#)
 - [event_key_soft_limit](#), [27](#)
 - [event_packet_index_limit](#), [27](#)
 - [event_ssrc_collision](#), [27](#)
 - [srtp_event_data_t](#), [26](#)
 - [srtp_event_handler_func_t](#), [26](#)
 - [srtp_event_t](#), [26](#)
 - [srtp_install_event_handler](#), [27](#)
- ssrc
 - [srtp_policy_t](#), [43](#)
- [ssrc_any_inbound](#)
 - SRTP, [17](#)
- [ssrc_any_outbound](#)
 - SRTP, [17](#)
- [ssrc_specific](#)

- SRTP, [17](#)
- ssrc_t, [44](#)
 - type, [44](#)
 - value, [44](#)
- ssrc_type_t
 - SRTP, [17](#)
- ssrc_undefined
 - SRTP, [17](#)
- stream
 - srtp_event_data_t, [41](#)
- STRONGHOLD_AUTH
 - Authentication, [33](#)
- STRONGHOLD_CIPHER
 - Ciphers, [30](#)
- type
 - ssrc_t, [44](#)
- UST_AES_128_XMAC
 - Authentication, [33](#)
- UST_TMMHv2
 - Authentication, [33](#)
- value
 - ssrc_t, [44](#)