libSRTP 1.4.4 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. Michael Jerris contributed support for building under MSVC. Andris Pavenis contributed many important fixes. Brian West contributed changes to enable dynamic linking. Yves Shumann reported documentation bugs. 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 Fredrik Thulin, Brian Weis, Mark Baugher, Jeff Chan, Bill Simon, Douglas Smith, Bill May, Richard Preistley, Joe Tardo and others for contributions, comments, and corrections.

This reference material in this documenation 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

1.8 ISMA Encryption Support 7

2 Module Index 9

2.1 Modules 9

3 Data Structure Index 11

3.1 Data Structures 11

4 Module Documentation **13**

4.1 Secure RTP 13

4.2 Secure RTCP 27

4.3 SRTP events and callbacks 29

4.4 Cryptographic Algorithms 32

4.5 Cipher Types 33

4.6 Authentication Function Types 36

4.7 Error Codes 38

4.8 Cryptographic Kernel 40

4.9 Ciphers 41

5 Data Structure Documentation **43**

5.1 crypto_policy_t Struct Reference 43

5.2 debug_module_t Struct Reference 45

5.3 srtp_event_data_t Struct Reference 46

5.4 srtp_policy_t Struct Reference 47

5.5 ssrc_t Struct Reference 50

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 approriate 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 libary, 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 extension[1] You probably want to get the most recent release. Unpack the distribution and extract the source files; the directory into which the soruce files will go is named `srtp`.

libSRTP uses the GNU `autoconf` and `make` utilities[2]. 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** diables stdout for error reporting.
- **–enable-console** use `/dev/console` for error reporting
- **–gdoi** use GDOI key management (disabled at present).

By default, dynamic debbuging 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).

---

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

[2] BSD 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
kernel_driver	crypto kernel (ciphers, auth funcs, rng)
srtp_driver	srtp in-memory tests (does not use the network)
rdbx_driver	rdbx (extended replay database)
roc_driver	extended sequence number functions
replay_driver	replay database
cipher_driver	ciphers
auth_driver	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 receve 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 avaliable 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 seperate 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 indepent 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 strucutre contains two crypto_policy_t structures that describe the cryptograhic 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;
uint8_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);
}
```

1.8 ISMA Encryption Support

The Internet Streaming Media Alliance (ISMA) specifies a way to pre-encrypt a media file prior to streaming. This method is an alternative to SRTP encryption, which is potentially useful when a particular media file will be streamed multiple times. The specification is available online at `http://www.isma.tv/specreq.nsf/SpecRequest.`

libSRTP provides the encryption and decryption functions needed for ISMAcryp in the library libaesicm.a, which is included in the default Makefile target. This library is used by the MPEG4IP project; see http://mpeg4ip.sourceforge.net/.

Note that ISMAcryp does not provide authentication for RTP nor RTCP, nor confidentiality for RTCP. ISMAcryp RECOMMENDS the use of SRTP message authentication for ISMAcryp streams while using ISMAcryp encryption to protect the media itself.

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

- Secure RTP 13
  - Secure RTCP 27
  - SRTP events and callbacks 29
- Cryptographic Algorithms 32
  - Cipher Types 33
  - Authentication Function Types 36
- Error Codes 38
- Cryptographic Kernel 40
  - Ciphers 41

Chapter 3

Data Structure Index

3.1 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) 43
debug_module_t 45
srtp_event_data_t (Srtp_event_data_t is the structure passed as a callback to the event handler function) . . 46
srtp_policy_t (Policy for an SRTP session) 47
ssrc_t (An ssrc_t represents a particular SSRC value, or a ‘wildcard’ SSRC) 50

Chapter 4

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.

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.

Modules

- Secure RTCP

  Secure RTCP functions are used to protect RTCP traffic.

- SRTP events and callbacks

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

Defines

- #define SRTP_MAX_TRAILER_LEN SRTP_MAX_TAG_LEN

  the maximum number of octets added by srtp_protect().

- #define crypto_policy_set_aes_cm_128_hmac_sha1_80(p) crypto_policy_set_rtp_default(p)

  crypto_policy_set_aes_cm_128_hmac_sha1_80() sets a crypto policy structure to the SRTP default policy for RTP protection.

Typedefs

- typedef struct ekt_policy_ctx_t ∗ ekt_policy_t

  points to an EKT policy

- typedef struct ekt_stream_ctx_t ∗ ekt_stream_t

  points to EKT stream data

- typedef struct srtp_ctx_t ∗ srtp_t

  An srtp_t points to an SRTP session structure.

- typedef struct 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 (void)

  srtp_init() initializes the srtp library.

- err_status_t srtp_shutdown (void)

  srtp_shutdown() de-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, unsigned int 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.

- void crypto_policy_set_aes_cm_128_hmac_sha1_32 (crypto_policy_t ∗p)

  crypto_policy_set_aes_cm_128_hmac_sha1_32() sets a crypto policy structure to a short-authentication tag policy

- void crypto_policy_set_aes_cm_128_null_auth (crypto_policy_t ∗p)

  crypto_policy_set_aes_cm_128_null_auth() sets a crypto policy structure to an encryption-only policy

- void crypto_policy_set_null_cipher_hmac_sha1_80 (crypto_policy_t ∗p)

  crypto_policy_set_null_cipher_hmac_sha1_80() sets a crypto policy structure to an authentication-only policy

- void crypto_policy_set_aes_cm_256_hmac_sha1_80 (crypto_policy_t ∗p)

  crypto_policy_set_aes_cm_256_hmac_sha1_80() sets a crypto policy structure to a encryption and authentication policy using AES-256 for RTP protection.

- void crypto_policy_set_aes_cm_256_hmac_sha1_32 (crypto_policy_t ∗p) *crypto_policy_set_aes_cm_256_hmac_sha1_32() sets a crypto policy structure to a short-authentication tag policy using AES-256 encryption.*

- err_status_t srtp_dealloc (srtp_t s)

  srtp_dealloc() deallocates storage for an SRTP session context.

- err_status_t crypto_policy_set_from_profile_for_rtp (crypto_policy_t ∗policy, srtp_profile_t profile)

  crypto_policy_set_from_profile_for_rtp() sets a crypto policy structure to the appropriate value for RTP based on an srtp_profile_t

- err_status_t crypto_policy_set_from_profile_for_rtcp (crypto_policy_t ∗policy, srtp_profile_t profile)

  crypto_policy_set_from_profile_for_rtcp() sets a crypto policy structure to the appropriate value for RTCP based on an srtp_profile_t

- unsigned int srtp_profile_get_master_key_length (srtp_profile_t profile)

  returns the master key length for a given SRTP profile

- unsigned int srtp_profile_get_master_salt_length (srtp_profile_t profile)

  returns the master salt length for a given SRTP profile

- void append_salt_to_key (unsigned char ∗key, unsigned int bytes_in_key, unsigned char ∗salt, unsigned int bytes_in_salt)

  appends the salt to the key

4.1.1 Define Documentation

4.1.1.1 #define crypto_policy_set_aes_cm_128_hmac_sha1_80(p) crypto_policy_set_rtp_default(p)

Parameters:

p is a pointer to the policy structure to be set

The function crypto_policy_set_aes_cm_128_hmac_sha1_80() is a synonym for crypto_policy_set_rtp_default(). It conforms to the naming convention used in RFC 4568 (SDP Security Descriptions for Media Streams).

Returns:

void.

4.1.1.2 #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().

4.1.2 Typedef Documentation

4.1.2.1 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.

4.1.2.2 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 intentially 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.

4.1.3 Enumeration Type Documentation

4.1.3.1 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).

Enumerator:

sec_serv_none	no services
sec_serv_conf	confidentiality

sec_serv_auth authentication

sec_serv_conf_and_auth confidentiality and authentication

4.1.3.2 enum ssrc_type_t

An ssrc_type_t enumeration is used to indicate a type of SSRC. See srtp_policy_t for more informataion.

Enumerator:

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())

4.1.4 Function Documentation

4.1.4.1 void append_salt_to_key (unsigned char ∗ *key*, unsigned int *bytes_in_key*, unsigned char ∗ *salt*, unsigned int *bytes_in_salt*)

The function call append_salt_to_key(k, klen, s, slen) copies the string s to the location at klen bytes following the location k.

Warning:

There must be at least bytes_in_salt + bytes_in_key bytes available at the location pointed to by key.

4.1.4.2 void crypto_policy_set_aes_cm_128_hmac_sha1_32 (crypto_policy_t ∗ *p*)

Parameters:

p is a pointer to the policy structure to be set

The function call crypto_policy_set_aes_cm_128_hmac_sha1_32(&p) sets the crypto_policy_t at location p to use policy AES_CM_128_HMAC_SHA1_32 as defined in RFC 4568. This policy uses AES-128 Counter Mode encryption and HMAC-SHA1 authentication, with an authentication tag that is only 32 bits long. This length is considered adequate only for protecting audio and video media that use a stateless playback function. See Section 7.5 of RFC 3711 (http://www.ietf.org/rfc/rfc3711.txt).

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.

Warning:

> This crypto policy is intended for use in SRTP, but not in SRTCP. It is recommended that a policy that uses longer authentication tags be used for SRTCP. See Section 7.5 of RFC 3711 (http://www.ietf.org/rfc/rfc3711.txt).

Returns:

> void.

4.1.4.3 void crypto_policy_set_aes_cm_128_null_auth (crypto_policy_t ∗ *p*)

Parameters:

> ***p*** is a pointer to the policy structure to be set

The function call crypto_policy_set_aes_cm_128_null_auth(&p) sets the crypto_policy_t at location p to use the SRTP default cipher (AES-128 Counter Mode), but to use no authentication method. This policy is NOT RECOMMENDED unless it is unavoidable; see Section 7.5 of RFC 3711 (http://www.ietf.org/rfc/rfc3711.txt).

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.

Warning:

> This policy is NOT RECOMMENDED for SRTP unless it is unavoidable, and it is NOT RECOMMENDED at all for SRTCP; see Section 7.5 of RFC 3711 (http://www.ietf.org/rfc/rfc3711.txt).

Returns:

> void.

4.1.4.4 void crypto_policy_set_aes_cm_256_hmac_sha1_32 (crypto_policy_t ∗ *p*)

Parameters:

p is a pointer to the policy structure to be set

The function call crypto_policy_set_aes_cm_256_hmac_sha1_32(&p) sets the crypto_policy_t at location p to use policy AES_CM_256_HMAC_SHA1_32 as defined in draft-ietf-avt-srtp-big-aes-03.txt. This policy uses AES-256 Counter Mode encryption and HMAC-SHA1 authentication, with an authentication tag that is only 32 bits long. This length is considered adequate only for protecting audio and video media that use a stateless playback function. See Section 7.5 of RFC 3711 (`http://www.ietf.org/rfc/rfc3711.txt`).

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.

Warning:

This crypto policy is intended for use in SRTP, but not in SRTCP. It is recommended that a policy that uses longer authentication tags be used for SRTCP. See Section 7.5 of RFC 3711 (`http://www.ietf.org/rfc/rfc3711.txt`).

Returns:

void.

4.1.4.5 void crypto_policy_set_aes_cm_256_hmac_sha1_80 (crypto_policy_t ∗ *p*)

Parameters:

p is a pointer to the policy structure to be set

The function call crypto_policy_set_aes_cm_256_hmac_sha1_80(&p) sets the crypto_policy_t at location p to use policy AES_CM_256_HMAC_SHA1_80 as defined in draft-ietf-avt-srtp-big-aes-03.txt. This policy uses AES-256 Counter Mode encryption and HMAC-SHA1 authentication, with an 80 bit authentication tag.

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.

4.1.4.6 err_status_t crypto_policy_set_from_profile_for_rtcp (crypto_policy_t ∗ *policy*, srtp_profile_t *profile*)

Parameters:

p is a pointer to the policy structure to be set

The function call crypto_policy_set_rtcp_default(&policy, profile) sets the crypto_policy_t at location policy to the policy for RTCP protection, as defined by the srtp_profile_t profile.

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:

values

- err_status_ok no problems were encountered
- err_status_bad_param the profile is not supported

4.1.4.7 err_status_t crypto_policy_set_from_profile_for_rtp (crypto_policy_t ∗ *policy*, srtp_profile_t *profile*)

Parameters:

p is a pointer to the policy structure to be set

The function call crypto_policy_set_rtp_default(&policy, profile) sets the crypto_policy_t at location policy to the policy for RTP protection, as defined by the srtp_profile_t profile.

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:

values

- err_status_ok no problems were encountered
- err_status_bad_param the profile is not supported

4.1.4.8 void crypto_policy_set_null_cipher_hmac_sha1_80 (crypto_policy_t * *p*)

Parameters:

p is a pointer to the policy structure to be set

The function call crypto_policy_set_null_cipher_hmac_sha1_80(&p) sets the crypto_policy_t at location p to use HMAC-SHA1 with an 80 bit authentication tag to provide message authentication, but to use no encryption. This policy is NOT RECOMMENDED for SRTP unless there is a requirement to forego encryption.

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.

Warning:

This policy is NOT RECOMMENDED for SRTP unless there is a requirement to forego encryption.

Returns:

void.

4.1.4.9 void crypto_policy_set_rtcp_default (crypto_policy_t * *p*)

Parameters:

p is a pointer to the policy structure to be set

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.

4.1.4.10 void crypto_policy_set_rtp_default (crypto_policy_t ∗ *p*)

Parameters:

p is a pointer to the policy structure to be set

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.

4.1.4.11 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 succeded.
- err_status_alloc_fail if stream allocation failed
- err_status_init_fail if stream initialization failed.

4.1.4.12 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. It may also be NULL, in which case streams should be added later using srtp_add_stream(). The final element of the list **must** have its 'next' field set to NULL.

Returns:

- err_status_ok if creation succeded.
- err_status_alloc_fail if allocation failed.
- err_status_init_fail if initialization failed.

4.1.4.13 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 occured.

4.1.4.14 err_status_t srtp_init (void)

Warning:

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

4.1.4.15 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

4.1.4.16 err_status_t srtp_remove_stream (srtp_t *session*, unsigned int *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 succeded.
- [other] otherwise.

4.1.4.17 err_status_t srtp_shutdown (void)

Warning:

No srtp functions may be called after calling this function.

4.1.4.18 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 ∗ *rtcp_hdr*, int ∗ *pkt_octet_len*)

The function call srtp_protect_rtcp(ctx, rtp_hdr, len_ptr) applies SRTCP protection to the RTCP packet rtcp_hdr (which has length ∗len_ptr) using the SRTP 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, srtp_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 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_ssrc_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 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.

Enumerator:

event_ssrc_collision	An SSRC collision occured.
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 fuction 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

- 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.

- 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_ICM 1

  AES Integer Counter Mode (AES ICM).

- #define AES_128_ICM AES_ICM

  AES-128 Integer Counter Mode (AES ICM) AES-128 ICM is a deprecated alternate name for AES ICM.

- #define SEAL 2

  SEAL 3.0.

- #define AES_CBC 3

  AES Cipher Block Chaining mode (AES CBC).

- #define AES_128_CBC AES_CBC

  AES-128 Cipher Block Chaining mode (AES CBC).

- #define STRONGHOLD_CIPHER AES_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 avaliable algorithms.

4.5.2 Define Documentation

4.5.2.1 #define AES_128_CBC AES_CBC

AES-128 CBC is a deprecated alternate name for AES CBC.

4.5.2.2 #define AES_CBC 3

AES CBC is the AES Cipher Block Chaining mode. This cipher uses a 16-, 24-, or 32-octet key.

4.5.2.3 #define AES_ICM 1

AES ICM is the variant of counter mode that is used by Secure RTP. This cipher uses a 16-, 24-, or 32-octet key concatenated with a 14-octet offset (or salt) value.

4.5.2.4 #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.

4.5.2.5 #define SEAL 2

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

4.5.2.6 #define STRONGHOLD_CIPHER AES_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 avaliable; 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 avaliable 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 avaliable; 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,
  err_status_socket_err = 16, err_status_signal_err = 17, err_status_nonce_bad = 18, err_status_read_fail = 19,
  err_status_write_fail = 20, err_status_parse_err = 21, err_status_encode_err = 22, err_status_semaphore_err = 23,
  err_status_pfkey_err = 24 }

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

Enumerator:

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

err_status_socket_err error in use of socket

err_status_signal_err error in use POSIX signals

err_status_nonce_bad nonce check failed

err_status_read_fail couldn’t read data

err_status_write_fail couldn’t write data

err_status_parse_err error pasring data

err_status_encode_err error encoding data

err_status_semaphore_err error while using semaphores

err_status_pfkey_err error while using pfkey

4.8 Cryptographic Kernel

Modules

- 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 uint8_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, uint8_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 Function Documentation

4.9.1.1 err_status_t cipher_dealloc (cipher_t ∗ *cipher*)

Warning:

May be implemented as a macro.

4.9.1.2 err_status_t cipher_encrypt (cipher_t $*$ *cipher*, void $*$ *buf*, unsigned int $*$ *len*)

Warning:

May be implemented as a macro.

4.9.1.3 err_status_t cipher_init (cipher_t $*$ *cipher*, const uint8_t $*$ *key*)

Warning:

May be implemented as a macro.

4.9.1.4 err_status_t cipher_output (cipher_t $*$ *c*, uint8_t $*$ *buffer*, int *num_octets_to_output*)

Warning:

May be implemented as a macro.

4.9.1.5 err_status_t cipher_set_iv (cipher_t $*$ *cipher*, void $*$ *iv*)

Warning:

May be implemented as a macro.

4.9.1.6 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

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 debug_module_t Struct Reference

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

- err.h

5.3 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

5.3.1 Detailed Description

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

5.3.2 Field Documentation

5.3.2.1 srtp_event_t srtp_event_data_t::event

An enum indicating the type of event.

5.3.2.2 srtp_t srtp_event_data_t::session

The session in which the event happend.

5.3.2.3 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

5.4 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
- unsigned char ∗ key
- ekt_policy_t ekt
- unsigned long window_size
- int allow_repeat_tx
- struct srtp_policy_t ∗ next

5.4.1 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 ‘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.

5.4.2 Field Documentation

5.4.2.1 int srtp_policy_t::allow_repeat_tx

Whether retransmissions of packets with the same sequence number are allowed. (Note that such repeated transmissions must have the same RTP payload, or a severe security weakness is introduced!)

5.4.2.2 ekt_policy_t srtp_policy_t::ekt

Pointer to the EKT policy structure for this stream (if any)

5.4.2.3 unsigned char∗ srtp_policy_t::key

Pointer to the SRTP master key for this stream.

5.4.2.4 struct srtp_policy_t∗ srtp_policy_t::next `[read]`

Pointer to next stream policy.

5.4.2.5 crypto_policy_t srtp_policy_t::rtcp

SRTCP crypto policy.

5.4.2.6 crypto_policy_t srtp_policy_t::rtp

SRTP crypto policy.

5.4.2.7 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.

5.4.2.8 unsigned long srtp_policy_t::window_size

The window size to use for replay protection.

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

- srtp.h

5.5 ssrc_t Struct Reference

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

Data Fields

- ssrc_type_t type
- unsigned int value

5.5.1 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).

5.5.2 Field Documentation

5.5.2.1 ssrc_type_t ssrc_t::type

The type of this particular SSRC

5.5.2.2 unsigned int 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, 34
AES_CBC
    Ciphers, 34
AES_ICM
    Ciphers, 34
allow_repeat_tx
    srtp_policy_t, 48
append_salt_to_key
    SRTP, 18
auth_key_len
    crypto_policy_t, 44
auth_tag_len
    crypto_policy_t, 44
auth_type
    crypto_policy_t, 44
auth_type_id_t
    Authentication, 37
Authentication
    auth_type_id_t, 37
    HMAC_SHA1, 37
    NULL_AUTH, 37
    STRONGHOLD_AUTH, 37
    UST_AES_128_XMAC, 37
    UST_TMMHv2, 37
Authentication Function Types, 36

Cipher Types, 33
cipher_dealloc
    CipherImplementations, 41
cipher_encrypt
    CipherImplementations, 41
cipher_init
    CipherImplementations, 42
cipher_key_len
    crypto_policy_t, 44
cipher_output
    CipherImplementations, 42
cipher_set_iv
    CipherImplementations, 42
cipher_type
    crypto_policy_t, 44
cipher_type_alloc
    CipherImplementations, 42
cipher_type_id_t
    Ciphers, 35
CipherImplementations
    cipher_dealloc, 41
    cipher_encrypt, 41
    cipher_init, 42
    cipher_output, 42
    cipher_set_iv, 42
    cipher_type_alloc, 42
Ciphers, 41
    AES_128_CBC, 34
    AES_CBC, 34
    AES_ICM, 34
    cipher_type_id_t, 35
    NULL_CIPHER, 34
    SEAL, 34
    STRONGHOLD_CIPHER, 34
crypto_policy_set_aes_cm_128_hmac_sha1_32
    SRTP, 18
crypto_policy_set_aes_cm_128_hmac_sha1_80
    SRTP, 16
crypto_policy_set_aes_cm_128_null_auth
    SRTP, 19
crypto_policy_set_aes_cm_256_hmac_sha1_32
    SRTP, 19
crypto_policy_set_aes_cm_256_hmac_sha1_80
    SRTP, 20
crypto_policy_set_from_profile_for_rtcp
    SRTP, 21
crypto_policy_set_from_profile_for_rtp

    SRTP, 21
crypto_policy_set_null_cipher_hmac_sha1_80
    SRTP, 22
crypto_policy_set_rtcp_default
    SRTP, 22
crypto_policy_set_rtp_default
    SRTP, 22
crypto_policy_t, 43
    auth_key_len, 44
    auth_tag_len, 44
    auth_type, 44
    cipher_key_len, 44
    cipher_type, 44
    sec_serv, 44
Cryptographic Algorithms, 32
Cryptographic Kernel, 40

debug_module_t, 45

ekt
    srtp_policy_t, 48
err_status_algo_fail
    Error, 39
err_status_alloc_fail
    Error, 38
err_status_auth_fail
    Error, 38
err_status_bad_param
    Error, 38
err_status_cant_check
    Error, 39
err_status_cipher_fail
    Error, 38
err_status_dealloc_fail
    Error, 38
err_status_encode_err
    Error, 39
err_status_fail
    Error, 38
err_status_init_fail
    Error, 38
err_status_key_expired
    Error, 39
err_status_no_ctx
    Error, 39
err_status_no_such_op
    Error, 39
err_status_nonce_bad
    Error, 39
err_status_ok
    Error, 38
err_status_parse_err
    Error, 39
err_status_pfkey_err
    Error, 39
err_status_read_fail
    Error, 39
err_status_replay_fail
    Error, 38
err_status_replay_old
    Error, 38
err_status_semaphore_err
    Error, 39
err_status_signal_err
    Error, 39
err_status_socket_err
    Error, 39
err_status_terminus
    Error, 38
err_status_write_fail
    Error, 39
err_status_t
    Error, 38
Error
    err_status_algo_fail, 39
    err_status_alloc_fail, 38
    err_status_auth_fail, 38
    err_status_bad_param, 38
    err_status_cant_check, 39
    err_status_cipher_fail, 38
    err_status_dealloc_fail, 38
    err_status_encode_err, 39
    err_status_fail, 38
    err_status_init_fail, 38
    err_status_key_expired, 39
    err_status_no_ctx, 39
    err_status_no_such_op, 39
    err_status_nonce_bad, 39
    err_status_ok, 38
    err_status_parse_err, 39
    err_status_pfkey_err, 39
    err_status_read_fail, 39

- err_status_replay_fail, 38
- err_status_replay_old, 38
- err_status_semaphore_err, 39
- err_status_signal_err, 39
- err_status_socket_err, 39
- err_status_terminus, 38
- err_status_write_fail, 39
- err_status_t, 38

Error Codes, 38
event
  srtp_event_data_t, 46
event_key_hard_limit
  SRTPevents, 30
event_key_soft_limit
  SRTPevents, 30
event_packet_index_limit
  SRTPevents, 30
event_ssrc_collision
  SRTPevents, 30

HMAC_SHA1
  Authentication, 37

key
  srtp_policy_t, 48

next
  srtp_policy_t, 48
NULL_AUTH
  Authentication, 37
NULL_CIPHER
  Ciphers, 34

rtcp
  srtp_policy_t, 48
rtp
  srtp_policy_t, 48

SEAL
  Ciphers, 34
sec_serv_auth
  SRTP, 17
sec_serv_conf
  SRTP, 17
sec_serv_conf_and_auth
  SRTP, 18
sec_serv_none
  SRTP, 17
sec_serv
  crypto_policy_t, 44
sec_serv_t
  SRTP, 17
Secure RTCP, 27
Secure RTP, 13
session
  srtp_event_data_t, 46
SRTCP
  srtp_protect_rtcp, 27
  srtp_unprotect_rtcp, 28
SRTP
  append_salt_to_key, 18
  crypto_policy_set_aes_cm_128_hmac_sha1_32, 18
  crypto_policy_set_aes_cm_128_hmac_sha1_80, 16
  crypto_policy_set_aes_cm_128_null_auth, 19
  crypto_policy_set_aes_cm_256_hmac_sha1_32, 19
  crypto_policy_set_aes_cm_256_hmac_sha1_80, 20
  crypto_policy_set_from_profile_for_rtcp, 21
  crypto_policy_set_from_profile_for_rtp, 21
  crypto_policy_set_null_cipher_hmac_sha1_80, 22
  crypto_policy_set_rtcp_default, 22
  crypto_policy_set_rtp_default, 22
  sec_serv_auth, 17
  sec_serv_conf, 17
  sec_serv_conf_and_auth, 18
  sec_serv_none, 17
  sec_serv_t, 17
  srtp_add_stream, 23
  srtp_create, 23
  srtp_dealloc, 24
  srtp_init, 24
  SRTP_MAX_TRAILER_LEN, 16
  srtp_protect, 24
  srtp_remove_stream, 25
  srtp_shutdown, 26
  srtp_stream_t, 17
  srtp_t, 17
  srtp_unprotect, 26
  ssrc_any_inbound, 18
  ssrc_any_outbound, 18
  ssrc_specific, 18
  ssrc_undefined, 18
  ssrc_type_t, 18
SRTP events and callbacks, 29

srtp_add_stream
  SRTP, 23
srtp_create
  SRTP, 23
srtp_dealloc
  SRTP, 24
srtp_event_data_t, 46
  event, 46
  session, 46
  stream, 46
srtp_event_handler_func_t
  SRTPevents, 30
srtp_event_t
  SRTPevents, 30
srtp_init
  SRTP, 24
srtp_install_event_handler
  SRTPevents, 31
SRTP_MAX_TRAILER_LEN
  SRTP, 16
srtp_policy_t, 47
  allow_repeat_tx, 48
  ekt, 48
  key, 48
  next, 48
  rtcp, 48
  rtp, 48
  ssrc, 48
  window_size, 48
srtp_protect
  SRTP, 24
srtp_protect_rtcp
  SRTCP, 27
srtp_remove_stream
  SRTP, 25
srtp_shutdown
  SRTP, 26
srtp_stream_t
  SRTP, 17
srtp_t
  SRTP, 17
srtp_unprotect
  SRTP, 26
srtp_unprotect_rtcp
  SRTCP, 28
SRTPevents
  event_key_hard_limit, 30
  event_key_soft_limit, 30
  event_packet_index_limit, 30
  event_ssrc_collision, 30
  srtp_event_handler_func_t, 30
  srtp_event_t, 30
  srtp_install_event_handler, 31
ssrc
  srtp_policy_t, 48
ssrc_any_inbound
  SRTP, 18
ssrc_any_outbound
  SRTP, 18
ssrc_specific
  SRTP, 18
ssrc_undefined
  SRTP, 18
ssrc_t, 50
  type, 50
  value, 50
ssrc_type_t
  SRTP, 18
stream
  srtp_event_data_t, 46
STRONGHOLD_AUTH
  Authentication, 37
STRONGHOLD_CIPHER
  Ciphers, 34

type
  ssrc_t, 50

UST_AES_128_XMAC
  Authentication, 37
UST_TMMHv2
  Authentication, 37

value
  ssrc_t, 50

window_size
  srtp_policy_t, 48