Appendix A: SyncP v2 Function Reference

Ford Motor Company

2.0.2

Contents

API Reference

SyncP

`initialize_syncp`

Name

`initialize_syncp` - Initialize the SyncP library.

Synopsis

```
#include <syncp/syncp_utils.h>
int initialize_syncp(void);
```

Return

`int`: Whether the initialize was successful, `0` on success, `1` on failure.

Parameters

`void`

Description

Initializes the SyncP library. This is required to be called prior to any other calls to SyncP. This will call OpenSSL's initialize function automatically, if compiled with OpenSSL.

`cleanup_syncp`

Name

`cleanup_syncp` - Deinitialize the SyncP library.

Synopsis

```
#include <syncp/syncp_utils.h>
int cleanup_syncp(void);
```

Return

`int`: Whether the deinitialization was successful, `0` on success, `1` on failure.

Parameters

`void`

Description

Deinitializes the SyncP library. Not required to be called, unless a graceful close of SyncP is required. This will call OpenSSL's cleanup function automatically, if compiled with OpenSSL.

`set_seed_gen`

Name

`set_seed_gen` - Set the seed generator

Synopsis

```
#include <syncp/syncp_utils.h>
int set_seed_gen(int (*crypto_gen_seed_in)(unsigned char *random_out, size_t num_bytes));
```

Return

`int`: Whether the function was successful, `0` on success, `1` on failure.

Parameters

`int (*crypto_gen_seed_in)(unsigned char *random_out, size_t num_bytes)`

The random seed generator function.

Description

This function is useful on microcontrollers where the usual ways to gain entropy (`/dev/random`, CPU timing uncertainty, etc) work terribly or are unavailable. `set_seed_gen` allows the application to provide a more secure random number generator (for ex. TPM, RNG, etc.) than TomCrypt or OpenSSL's built-in generators. The provided function is only called when the pseudorandom number generator needs seeding or reseeding.

`get_rand_bytes`

Name

`get_rand_bytes` - Gets pseudorandom bytes

Synopsis

```
#include <syncp/syncp_utils.h>
int get_rand_bytes(unsigned char *buff, size_t len);
```

Return

`int`: Whether the function was successful, `0` on success, `1` on failure.

Parameters

`unsigned char *buff`

The buffer to put random data

`size_t len`

The number of bytes to generate

Description

Gets random data from the underlying cryptogrpahic library's PRNG. Note: this function does not directly call a seed function if provided from `set_seed_gen`, but may cause a call if the internal PRNG is determined to need a reseed.

`memcmp_consttime`

Name

`memcmp_consttime` - Compare memory regions in constant time

Synopsis

```
#include <syncp/syncp_utils.h>
int memcmp_consttime(const unsigned char *buf1, const unsigned char *buf2, size_t size);
```

Return

`int`: Whether the regions of memory are the same, `0` if equal, nonzero if not.

Parameters

`const unsigned char *buf1`

The region of memory

`const unsigned char *buf1`

The region of memory to compare with

`size_t size`

The number of bytes to compare

Description

This function acts like `memcmp`, except it will process all `size` bytes of the input, rather than exiting if a difference is found. This function is useful when attempting to avoid timing attacks on secret material. It should not be used, and may be inefficient for general memory comparison.

---

get_encoded_size

Name

`get_encoded_size` - Gets the encoded size of a packet

Synopsis

```
#include <syncp/syncp_utils.h>
size_t get_encoded_size(const syncp_packet *packet);
```

Return

`size_t`: The size in bytes. Zero on error.

Parameters

`const syncp_packet *packet`

The packet to encode, as written by `encode_init` (`v0` or `v1`)

Description

Takes a packet written by `encode_init` (`v0` or `v1`) and will return the number of bytes required to sufficiently store the encoded packet. This may return a few bytes more than actually used, but is generally a tight upper bound on required buffer size. This is useful for determining how many bytes should be allocated to process an encode, or to check whether a fixed buffer is large enough to store the desired resulting packet.

---

get_crypto

Name

`get_crypto` - Gets a cryptolibrary reference

Synopsis

```
#include <syncp/syncp_utils.h>
const syncp_crypto_library * get_crypto(library_t lib);
```

Return

`const syncp_crypto_library *`: A pointer to the specified cryptolibrary, or `nullptr` if not available.

Parameters

`library_t lib`

The library to retrieve, defined in

Description

Gets a reference to the specified cryptolibrary. This function is only really useful if both TomCrypt and OpenSSL support is compiled in, and the application needs to switch between the two. The return value of this function should be passed to `replace_crypto` in order to swap the active cryptolibrary.

---

replace_crypto

Name

`replace_crypto` - Replaces the current cryptolibrary

Synopsis

```
#include <syncp/syncp_utils.h>
int replace_crypto(const syncp_crypto_library *lib);
```

Return

`int`: Whether the function was successful, `0` on success, `1` on failure.

Parameters

`const syncp_crypto_library *lib`

The library to set active.

Description

Sets the active cryptolibrary to the library specified. The parameter should be a cryptolibrary that is retreived via `get_crypto`.

`set_passback_data`

Name

`set_passback_data` - Sets arbitrary passback data for the key provider implementation to reference.

Synopsis

```
#include <syncp/syncp_utils.h>
int set_passback_data(syncp_packet *packet, void *data);
```

Return

`int`: Whether attaching the data was successful, `0` on success, `1` on failure.

Parameters

`syncp_packet *packet`

The packet to which the passback data should be attached.

`void *data`

The data to attach.

Description

Allows application or configuration data to be passed through to the key provider function for easy lookup. This is useful, for example, if the key provider implementation requires an ESN, VIN, or other piece of data in order to access the correct key. Since the `set_passback_data` function requires an initialized packet, it must be called after either the `encode_init` or `decode_init` functions are called.

`encode_init`

Name

`encode_init` - Initializes a packet structure for encoding

Synopsis

```
#include <syncp/v1/v1_syncp_utils.h>
int encode_init(
    syncp_packet        *packet,
    unsigned int         has_message_id,
    unsigned int         has_esn,
    unsigned int         has_key_id,
    unsigned int         cpu_destination,
    unsigned short       service_type,
    unsigned short       command_type,
    unsigned char        cryptotype,
```

```
    const unsigned char *plaintext,
    unsigned long        plaintext_size,
    _sp_esn_t           *esn,
    unsigned char        key_sign_slot,
    unsigned char        key_encrypt_slot,
    uint64_t             module_message_id,
    uint64_t             server_message_id,
    unsigned char        message_status
);
```

Return

`int`: Whether the function was successful, `0` on success, `1` on failure.

Parameters

`syncp_packet *packet`

A pointer to the packet needing initializing

`unsigned int has_message_id`

`1` or `0`, whether `module_message_id`, `server_message_id`, and `message_status` will be included in the packet.

`unsigned int has_esn`

`1` or `0`, whether an `esn` value will be included in the packet.

`unsigned int has_key_id`

`1` or `0`, whether a `key_id` value will be included in the packet.

`unsigned int cpu_destination`

The CPU destination of the packet.

`unsigned short service_type`

The service type for the packet

`unsigned short command_type`

The command type for the packet

`unsigned char cryptotype`

The cryptographic operation to use when encoding the packet

`const unsigned char *plaintext`

The message to send

`unsigned long plaintext_size`

The size of the message

`_sp_esn_t *esn`

The ESN of the module

`unsigned char key_sign_slot`

The signing key's slot (ignored for authenticated encryption)

`unsigned char key_encrypt_slot`

The encryption key's slot. This field is ignored if `cryptotype` is set as an authenticated encryption mode (CCM, GCM)

`unsigned long module_message_id`

Module Message ID. This field is ignored if `message_id_reqired` is `0`

`unsigned long server_message_id`

Server Message ID. This field is ignored if `message_id_reqired` is `0`

`unsigned char message_status`

Message Status. This field is ignored if `message_id_reqired` is `0`

Description

The `encode_init` function prepares a `syncp_packet` structure for encoding, validating and setting the various fields on the packet to prepare for the encoding. Since several fields in the packet protocol are not required in all cases, some parameters may not be used. In these cases, it is safe to insert a null or zero value for any unused fields. Special care should be exercised with the key slot identifiers, as the `cryptotype` changes which keys will be used. The first key is used in all cases except authenticated encryption modes, wheras the second is only used in authenticated encryption modes and asymmetric modes that perform both encryption and signing.

`v0_encode_init`

Name

`v0_encode_init` - Initializes a v0 packet structure for encoding

Synopsis

```
#include <syncp/v0/v0_syncp_utils.h>
int v0_encode_init(
    syncp_packet        *packet,
    unsigned char        response_required,
    unsigned char        cryptotype,
    unsigned char        has_esn,
    unsigned char        service_type,
    unsigned char        command_type,
    unsigned char        cpu_destination,
    unsigned char        key_slot,
    const unsigned char *plaintext,
    unsigned long        plaintext_size,
    _sp_esn_t           *esn,
    unsigned long        module_message_id,
    unsigned long        server_message_id,
    unsigned char        message_status
);
```

Return

`int`: Whether the function was successful, `0` on success, `1` on failure.

Parameters

`syncp_packet *packet`

A pointer to the packet needing initializing

`unsigned char response_required`

`1` or `0`, whether `module_message_id` and `server_message_id` will be included.

`unsigned char cryptotype`

The cryptographic operation to use when encoding the packet

`unsigned char has_esn`

`1` or `0`, whether an `esn` value will be included.

`unsigned short service_type`

The service type for the packet

`unsigned short command_type`

The command type for the packet

`unsigned int cpu_destination`

The CPU destination of the packet.

`unsigned char key_slot`

The slot of the encryption/signing key used to encode the packet

`const unsigned char *plaintext`

The message to send

`unsigned long plaintext_size`

The size of the message

`_sp_esn_t *esn`

The ESN of the module

`unsigned long module_message_id`

Module Message ID. This field is ignored if `message_id_reqired` is `0`

`unsigned long server_message_id`

Server Message ID. This field is ignored if `message_id_reqired` is `0`

`unsigned char message_status`

Message Status. This field is ignored if `message_id_reqired` is `0`

Description

The `v0_encode_init` function prepares a `syncp_packet` structure for encoding, validating and setting the various fields on the packet to prepare for the encoding. Since several fields in the packet protocol are not required in all cases, some parameters may not be used. In these cases, it is safe to insert a null or zero value for any unused fields. Special care should be exercised with the key slot identifiers, as the `cryptotype` changes which keys will be used. The first key is used in all cases except authenticated encryption modes, wheras the second is only used in authenticated encryption modes and asymmetric modes that perform both encryption and signing.

`decode_init`

Name

`decode_init` - Initializes a packet structure for decoding

Synopsis

```
#include <syncp/syncp_utils.h>
int decode_init(syncp_packet *packet, const unsigned char *buffer, size_t buffer_size);
```

Return

`int`: Whether the function was successful, `0` on success, `1` on failure.

Parameters

`syncp_packet *packet`

The packet structure to initialize for decoding

`const unsigned char *buffer`

The buffer where the raw packet data resides

`size_t buffer_size`

The size in bytes of the packet data buffer

Description

`decode_init` prepares a `syncp_packet` structure for decoding, reading header information from the packet and setting appropriate fields within the packet structure (such as cryptotype, required keys, etc) for decoding.

`encode_packet`

Name

`encode_packet` - Encodes a SyncP packet

Synopsis

```
#include <syncp/syncp_utils.h>
int encode_packet(const syncp_packet *packet, unsigned char *buffer, size_t *buffer_size);
```

Return

`int`: Whether the function was successful, `0` on success, `1` on failure.

Parameters

`const syncp_packet *packet`

The packet to encode, initialized by `encode_init`

`unsigned char *buffer`

The buffer to write the encoded packet into

`size_t *buffer_size`

The number of bytes available to write to the buffer. After the encode, this variable will be changed to the number of bytes written to the buffer, indicating the packet size.

Description

`encode_packet` takes an initialized packet structure, performs all encoding, encryption, signing key resolution, etc. as necessary, and writes the result to `buffer`, updating `buffer_size` to the number of bytes written.

`decode_packet`

Name

`decode_packet` - Decodes a SyncP packet

Synopsis

```
#include <syncp/syncp_utils.h>
int decode_packet(syncp_packet *packet, unsigned char *buffer, size_t buffer_size);
```

Return

`int`: Whether the function was successful, `0` on success, `1` on failure.

Parameters

`syncp_packet *packet`

The packet to decode, initialized by `decode_init`

`unsigned char *buffer`

The buffer to write the decoded plaintext into

`size_t buffer_size`

The size of the plaintext buffer.

Description

`decode_packet` takes an initialized packet structure, performs all decoding, decryption, verification key resolution, etc. as necessary, and writes the metadata to `packet` and plaintext to `buffer`

Key Providers

`set_TYPE_key_provider`

Name

`set_TYPE_key_provider` - Sets a key provider

Synopsis

```
#include <syncp/key_store.h>
void set_sym128_key_provider      (sym128_key_provider      provider);
void set_sym256_key_provider      (sym256_key_provider      provider);
void set_rsa_private_key_provider (rsa_private_key_provider provider);
void set_rsa_public_key_provider  (rsa_public_key_provider  provider);
void set_ec_private_key_provider  (ec_private_key_provider  provider);
void set_ec_public_key_provider   (ec_public_key_provider   provider);
```

Return

`void`

Parameters

`TYPE_key_provider provider`

A pointer to a keyprovider of type TYPE. Each key provider accepts a buffer to write the key material to as well as which key to use, via the key slot identifier.

Description

`set_TYPE_key_provider` sets the function that will be called when SyncP needs to load a key of type `TYPE`. For example, if a packet is being process with a cryptotype of `CRYPTOTYPE_SYM_SIGN_128_CMAC_PBKDF2`, the `sym128_key_provider` function will be called. If a provider of this type has not been set via it's associated set function, the processing will fail, and SyncP will return an error.

Note, this function may be called at any time to change the currently set key provider, which may be useful if the device continues to operate while an upgrade or key rotation is in place. Passing `nullptr` to a function of this type will clear the set key provider, and will cause SyncP to abort encoding and decoding of packets requiring a key of that type.

`get_TYPE_key_provider`

Name

`get_TYPE_key_provider` - Gets a key provider

Synopsis

```
#include <syncp/key_store.h>
sym128_key_provider      get_sym128_key_provider(void);
sym256_key_provider      get_sym256_key_provider(void);
rsa_private_key_provider get_rsa_private_key_provider(void);
rsa_public_key_provider  get_rsa_public_key_provider(void);
ec_private_key_provider  get_ec_private_key_provider(void);
ec_public_key_provider   get_ec_public_key_provider(void);
```

Return

`TYPE_key_provider`: A function pointer to the key provider, or `nullptr` if one has not been set.

Parameters

`(void)`

Description

`get_TYPE_key_provider` retrieves the currently set key provider of type `TYPE`. This may be useful if the device continues to operate while an upgrade or key rotation is in place to check if the correct key provider is set (or if one is set at all).

Logging

`set_log_level`

Name

`set_log_level` - Sets the log level

Synopsis

```
#include <syncp/log.h>
void set_log_level(log_level level);
```

Return

`void`

Parameters

`log_level level`

The lowest level of messages to log.

Description

`set_log_level` sets the minimum severity on log messages. For example, if `LOG_LEVEL_WARNING` is set, informational and debugging information will not be logged - only messages at least as severe as warnings are logged.

Note: if `DEVELOP` is not defined, `set_log_level` will compile to `(void) 0`.

`set_log_location`

Name

`set_log_location` - Sets the location that logs are written to

Synopsis

```
#include <syncp/log.h>
void set_log_location(FILE *log);
```

Return

`void`

Parameters

`FILE *log`

A file handle to write the logs to

Description

`set_log_location` sets the file handle to write logs to. By default, this is set to stderr.

Note: if `DEVELOP` is not defined, `set_log_location` will compile to `(void) 0`.

`log_print` (macro)

Name

`log_print` - Prints a message to the log

Synopsis

```
#include <syncp/log.h>
#define log_print(criticality, ...) log_print_extra(0, 0, criticality, __VA_ARGS__)
```

Return

`void`

Parameters

`log_level criticality`

The severity of the message.

`const char *fmt`

The message format, which is the same as `printf`'s format string.

`...`

Any extra arguments, as specified by the format string.

Description

`log_print` prints a message, prefixed by the message severity to the logs. Internally, this calls to `printf`, and supports all of `printf`'s formats and arguments. `log_print` adds a newline to the end of every log line.

Note: if `DEVELOP` is not defined, `log_print` will compile to `(void) 0`.

`status_message` (macro)

Name

`status_message` - Prints a message to the log, and returns a status code

Synopsis

```
#include <syncp/log.h>
#define status_message(status, criticality, ...) \
      (log_print_extra(__LINE__, __FILE__, criticality, __VA_ARGS__), status)
```

Return

`void`

Parameters

`int status`

The resulting value from the operation, acting like a return value.

`log_level criticality`

The severity of the message.

`const char *fmt`

The message format, which is the same as `printf`'s format string.

`...`

Any extra arguments, as specified by the format string.

Description

`status_message` prints a message, prefixed with the message severity, filename, and line number to the logs. Internally, this calls to `printf`, and supports all of `printf`'s formats and arguments. `status_message` adds a newline to the end of every log line. Additionally, `status_message` leaves a resulting value from the macro, allowing this macro to be used in a return statement, like so:

```
int main(int argc, char **argv)
{
    return status_message(1, LOG_LEVEL_ERROR, "%d arguments passed.", argc);
}
```

This will result in an exit value of 1, as well as the following log line:

```
[ERROR] main.c:3 | 1 arguments passed.
```

Note: if `DEVELOP` is not defined, `status_message` will compile to `((void) 0, status)`.

`enable_log`

Name

`enable_log` - Enables the log

Synopsis

```
#include <syncp/log.h>
void enable_log(void);
```

Return

`void`

Parameters

`void`

Description

`enable_log` enables the log, if it has been disabled. By default, the log is enabled if `DEVELOP` is defined.

Note: if `DEVELOP` is not defined, `set_log_location` will compile to `(void) 0`.

`disable_log`

Name

`disable_log` - Disables the log

Synopsis

```
#include <syncp/log.h>
void disable_log(void);
```

Return

`void`

Parameters

`void`

Description

`disable_log` disables all logging activity.

Note: if `DEVELOP` is not defined, `set_log_location` will compile to `(void) 0`.