SyncP Version 2 Developer Guide

Ford Motor Company

2.0.2

Contents

About

This is the second version of the SyncP library, which implements version 1 of the SyncP protocol. To help alleviate confusion, the SyncP library will be referred to as "SyncP", wheras the SyncP protocol will be referred to as "Protocol"

SyncP v2 is not API or ABI compatable with the SyncP v1, however support for encoding and decoding some Protocol v0 packets (excepting low bandwidth mode and encryption only).

This updated version of SyncP contains many enhacements over the first version, including:

- OpenSSL Compatability
- Greater range of crypto-operations (Elliptic Curve, PBKDF2 Deriviation, etc.)
- Greater flexibility with Pluggable Cryptographic Modules
- Extensive unit testing
- Standard *nix build (`./configure.sh make`), with `nmake` compatibility on windows.
- And much more.

In addition, it is written in portable ISO-C99 following the MisraC3 2012 specification. It is designed to integrate with application code in a wide variety of circumstances with minimal modification. However, this also serves as a reference implementation for applications where it is necessary to rewrite portions or all of the library.

SyncP is single-threaded by design. While it may be used in a multi-threaded architecture, it is important to only access SyncP functionality from a single thread.

This Developer Guide covers only this reference implementation. Custom implementations are only required to adhere to the Protocol Specification.

You should have received the SyncP Protocol Specification along with the SyncP Source and Developer Guide.

Ford Confidential

Applicability

The code contained here, in addition to all associated documentation, notes, clarifications, etc. are subject to the above confidenential status.

Code Layout

Inside of the `/src/` directory, there are several different sections in the SyncP code reposatory. Each of these will be explained in greater detail below.

- `crypt/`
  - Contains all of the cryptographic interface code that is used by syncp. Application code should not reference anything inside `crypt/`
- `syncp/`
  - The core SyncP interface code, wraps cryptocode into an easy-to-use API.
- `examples/`
  - Contains simple examples to help illustrate how the SyncP version 2 library is used
- `tests/`
  - Contains the test cases that ensure SyncP works as expected.
- `TomCrypt/`

    - A slightly modified version of TomCrypt, to eliminate dynamic memory allocation.
- `TomsFastMath/`
    - A slightly modified version of TomsFastMath, to eliminated dynamic memory allocation.

`crypt/`

Inside `crypt/` is all of the core cryptographic operations that allow syncp to operate. Since the call path to use these operations will differ between cryptolibraries (TomCrypt, OpenSSL, etc.), this code makes all necessary calls out to the appropriate cryptographic library, and presents a consistant interface to SyncP, which enables the ability to easily create pluggable cryptographic modules. These modules may be swapped at runtime, via the loading of function pointers to each primitive operation. The list of these operations, the function pointer typedefs, the library structure, and global function pointer declarations can be found in `crypt/cryptofunctions.h`

Some operations are not strictly “primitive”, but are a composition of primitives and act together. An example of this is `rsa_envelope_seal`, which performs a `symmetric_encrypt` primitive operation followed by an `rsa_encrypt` primitive operation of the symmetric key used. Declarations for these composite operations can also be found in `crypt/cryptofunctions.h`, and the definitions in `crypt/cryptofunctions.c`

In addition to `cyptofunctions`, there are two crypto-libraries provided by default: OpenSSL and TomCrypt. Each library implements the full gamut of cryptographic primitives, and additional libraries here are expected to do the same.

If desired, the code in `crypt/` may be built into a dynamic library separate from SyncP.

`syncp/`

`syncp/` contains all of the core SyncP code. This includes packet network endianness translation, encoding/decoding, module management, key handling, debug logging, and API. There are two main folders within `syncp/`, `v0`, and `v1`, for encoding/decoding protocol version 0 and protocol version 1 packets respectively. The layout of code is the same between these two directories.

Functions for encoding packet fields are defined in `syncp/vx/fields/vxfields_encode_decode.h`, with the implementation split across multiple files in `syncp/fields/*.c`. Encoding and decoding the packet base is handled in `syncp/v1/vx_base.h` and `syncp/vx_base.c`. The functions to perform the encryption and decryption, as well as encoding and decoding of cryptotype groups are defined in `syncp/vx/vx_syncp_encode_decode.h`, and are split across several files according to their group:

- `v1_none.c`
- `v1_sym_sign.c`
- `v1_sym_encrypt_auth.c`
- `v1_asym_sign.c`
- `v1_rsa_encrypt_sign.c`
- `v1_ec_encrypt_sign.c`
- `v1_rsa_envelope_sign.c`

And

- `v0_none.c`
- `v0_sym_encrypt_auth.c`
- `v0_sym_sign.c`

There are several helper functions declared in `cryptohelper.h`, which are intended for internal use in the SyncP library. These functions derive various parameters from the `cryptotype` field.

Several files are intended to be included by application code:

- `syncp.h` defines cryptotype constants and field types
- `v0/v0_syncp.h` defines cryptotype constants and field types for protocol v0
- `v1/v1_syncp.h` defines cryptotype constants and field types for protocol v1
- `syncp_utils.h` declares the API functions for creating, encoding, and decoding SyncP packets.
- `v0/v0_syncp_utils.h` declares the API functions for creating, encoding, and decoding version 0 SyncP packets.

- `v1/v1_syncp_utils.h` declares the API functions for creating, encoding, and decoding version 0 SyncP packets.
- `key_store.h` declares the functions for setting up key providers
- `data.h` includes functions for encoding and decoding various integer sizes such that transfer occurs in network order.
- `log.h` defines functions for controlling logging behavior if `DEVELOP` mode is enabled.

`examples/`

This folder contains simple example applications that integrate SyncP. See `Makefile` to see how to build these examples.

`tests/`

A repository of test cases. These tests function as complete applications that test various functionality within the SyncP library code. These tests can also be referred to as additional example code.

`TomCrypt/` and `TomsFastMath/`

These directories contain the code for TomCrypt and TomsFastMath, modified in such a way as to bypass all dynamic allocation and assembly code. TomsFastMath was used instead of the standard LibTomMath due to memory allocation and portability concerns. Modifications to these libraries are documented in `TomCrypt Source Changes.txt`, however only modifications deemed necessary were performed. Given this is a large external library, it does not necessarily pass all Misra rules.

Note: If `crypt/` is built into its own libcryptofunctions dynamic library, this will staticly embed this customized version of tomcrypt into the library.

Getting Started

A Simple Example

First off, `syncp.h` and `syncp_utils.h` need to be included, `v1_syncp_utils.h` because this example shows encoding of v1 packets. `key_store.h` will be required to actually do any cryptographic operations, however this example uses the `NONE` cryptotype, which will only perform a raw packet encoding. Key providers are explained in more details later, so it is omitted here.

```
#include <string.h>
#include <syncp/syncp.h>
#include <syncp/syncp_utils.h>
#include <syncp/v1/v1_syncp_utils.h>

int main()
{
```

Next, the library needs to be initialized. Error handling is being intentionally ignored here in the interest of simplicity, but the examples in `examples/` show more detail in this regard.

```
    initialize_syncp();
```

Now, each parameter for the encoding process gets defined.

```
    syncp_packet   packet;
    unsigned int   has_message_id      = 0;
    unsigned int   has_esn             = 0;
    unsigned int   has_key_id          = 1;
    unsigned int   cpu_destination     = 0;
    unsigned short service_type        = 0;
    unsigned short command_type        = 0;
    unsigned short cryptotype          = CRYPTOTYPE_NONE;
```

```
        const unsigned char *plaintext = "Hello World";
        unsigned long plaintext_length = strlen(message);
```

Now that all of our parameters are set, the packet may now be prepared for encoding. Here, `encode_init` will take the parameters specified, and will populate the `packet` struture, validating the fields passed in. Since no ESN, Message IDs, or keys, are being set, there are several empty values at the end. Any parameter that is not needed is simply ignored.

```
        encode_init(
            &packet,
            has_message_id,
            has_esn,
            has_key_id,
            cpu_destination,
            service_type,
            command_type,
            cryptotype,
            plaintext,
            plaintext_length,
            nullptr,
            0, 0,
            0, 0, 0
        );
```

Now that the packet is initialized, encoding can proceed. Here, a buffer is passed in which will hold the encoded packet data. It is recommended to check the size of the buffer via `get_encoded_size`, however the buffer here is plenty large to store the packet.

```
        unsigned char packet_buffer[256];
        size_t buffer_size = 256;
        encode_packet(&packet, packet_buffer, &buffer_size);
```

Here, `encode_packet` will take the packet, check that there is sufficient space in the buffer, then encode and write the size of the buffer back to `buffer_size`. From here, `packet_buffer` may be saved to a file, sent across the network, etc. In this example, a decode operation is performed to recover the original plaintext. To do this, much like an encode operation, a `decode_init` call is made on a new `syncp_packet`.

```
        syncp_packet packet_decode;
        decode_init(&packet_decode, packet_buffer, buffer_size);
```

Next, a block of memory is allocated to hold the plaintext, followed by the decode operation. Again, it is recommended to check the size of the buffer through `packet_decode.plaintext_size`, however for simplicity, this step is ommitted.

```
        unsigned char decoded_plaintext[64];
        decode_packet(&packet_decode, decoded_plaintext, 64);
```

Now, `decoded_plaintext` holds the original plaintext of `"Hello World"`.

```
        return 0;
    }
```

Key Providers

One of the major benefits of SyncP v2 is its expanded support for cryptographic operations. Using `CRYPTOTYPE_NONE` for all packets is insecure, since no message authentication or encryption can be performed, and also defeats one of the major features of SyncP - security! Thus, in order be able to use cryptography in SyncP, it is necessary to tell SyncP how to access keys when needed for encoding and decoding.

The Key Provider system is designed to automatically resolve needed keys as well as their types. As keys may be stored in device-specific locations, SyncP cannot access these directly. Instead, application will define a method by which SyncP can access the keys, which is then called as needed. Because SyncP can now handle passing keys through the library, it greatly simplifies the encoding and decoding processes, as it is no longer necessary to peak at the packet headers to determine which key to use.

Key Provider Types

There are six key provider types, as defined in `syncp/key_store.h`:

```
typedef size_t (*sym128_key_provider)
    (sym128_key_t *, _sp_key_id_t *, unsigned, void *);

typedef size_t (*sym256_key_provider)
    (sym256_key_t *, _sp_key_id_t *, unsigned, void *);

typedef size_t (*rsa_private_key_provider)
    (rsa_private_key_t *, _sp_key_id_t *, unsigned, void *);

typedef size_t (*rsa_public_key_provider)
    (rsa_public_key_t *, _sp_key_id_t *, unsigned, void *);

typedef size_t (*ec_private_key_provider)
    (ec_private_key_t *, _sp_key_id_t *, unsigned, void *);

typedef size_t (*ec_public_key_provider)
    (ec_public_key_t *, _sp_key_id_t *, unsigned, void *);
```

Each key provider is a function which accepts a pointer to stack memory in which to write a key, as well the Key Slot which identifies the key on the device. The value returned should be the number of bytes written in to `*keybuf`, or zero on error. A `nullptr` may be passed to the keyprovider function. In this case, the size of the key that would have been copied shall be returned.

`sym128_key_t` and `sym256_key_t` are 16 and 32-byte blocks of memory, respectively. `rsa_private_key_t` and `rsa_public_key_t` are sized to hold 2048-bit PKCS8-formatted, DER-encoded keys. Similarly, `ec_private_key_t` and `ec_public_key_t` are sized to hold PKCS8-formatted, DER-encoded EC keys.

Setting a key provider

There are six functions to set the providers of each key type. These will usually be called within the application initialization procedure, but may be called at any time to update or change the key provider functionality. These functions may be changed multiple times throughout the application lifecycle, if necessary. This is a great way to inject fake keys for testing the crypto platform, or for debugging a misbehaving application.

Omitting a provider, or calling the function to set the provider to null will inform SyncP that a particular type of key is not available, or is no longer available. Upon being requested to decode or encode a packet with a missing key, SyncP will return an error.

```
void set_sym128_key_provider(sym128_key_provider);
void set_sym256_key_provider(sym256_key_provider);
void set_rsa_private_key_provider(rsa_private_key_provider);
void set_rsa_public_key_provider(rsa_public_key_provider);
void set_ec_private_key_provider(ec_private_key_provider);
void set_ec_public_key_provider(ec_public_key_provider);
```

All that is needed to do to set a key provider is to define a key provider function matching a key provider function pointer type, then call it's associated set function:

```
size_t key_provider(sym128_key_t *key, _sp_key_id_t *key_id, unsigned key_index, void *pb)
{
    // Code to copy key to *key, if not null

    return 16;
}
```

Followed by a `set_sym128_key_provider(key_provider_example)` on application start.

Using Key Providers

This example builds upon the previous example. For a complete, working example, please refer to the example code in `examples/sym_codec.c`. In this example, a symmetric key is hardcoded into the application code, however the actual storage location is irrelevant from the perspective of SyncP.

First, a key provider method is defined:

```
    #include <syncp/syncp.h>
    #include <syncp/syncp_utils.h>
    #include <syncp/v1/v1_syncp_utils.h>
    #include <syncp/key_store.h>

    size_t key_provider_k128(sym128_key_t *key, _sp_key_id_t *key_id, unsigned key_index, void *pb)
    {
        // Ignore the Key ID - let SyncP generate this
        (void) key_id;

        // Ignore the Passback Data
        (void) pb;

        // Hardcoded key
        unsigned char sym_key[16] =
        {
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
            0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
        };

        // Return 0 on error (requested key does not exist)
        if (key_index != 0)
            return 0;

        // Copy key into key buffer, if provided
        if (key)
            memcpy(key, sym_key, sizeof(sym_key));

        // Return number of bytes of the key
        return sizeof(sym_key);
    }
```

Then, in the application code, this can now be set as a key provider:

```
    int main()
    {
        initialize_syncp();
        set_sym128_key_provider(key_provider_k128);
```

Any encode or decode calls from this point forward, which require a 128-bit symmetric key will call `key_provider_k128` to obtain the key automatically. To continue the example:

```
        syncp_packet   packet;
        unsigned int   has_message_id      = 0;
        unsigned int   has_esn             = 0;
        unsigned int   has_key_id          = 1;
        unsigned int   cpu_destination     = 0;
        unsigned short service_type        = 0;
        unsigned short command_type        = 0;
        unsigned short cryptotype          = CRYPTOTYPE_SYM_SIGN_128_HMAC_PBKDF2;
        unsigned short key_sign_slot       = 0;

        const unsigned char *plaintext = "Hello World";
        unsigned long plaintext_length = strlen(message);

        encode_init(
            &packet,
            has_message_id,
            has_esn,
            has_key_id,
            cpu_destination,
            service_type,
            command_type,
            cryptotype,
            plaintext,
            plaintext_length,
            nullptr,
            key_sign_slot,
            0, 0, 0, 0
        );
```

From there, the example continues the same as before.

```
        unsigned char packet_buffer[256];
        size_t buffer_size = 256;
        packet_encode(&packet, packet_buffer, &buffer_size);
```

Key providers really shine when it comes to decoding SyncP packets, since it is no longer necessary to peek at the header to load the right key. SyncP will do this automatically.

```
        syncp_packet decode;
        unsigned char decoded_plaintext[64];

        decode_init(&packet_decode, packet_buffer, buffer_size);
        decode_packet(&packet_decode, decoded_plaintext, 64);

        return 0;
    }
```

Passback Data

Passback data allows the application to carry configuration data 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. By default, this pointer is null, however it can be set with a call to `set_passback_data(syncp_packet *packet, void *data)`. 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.

Because the data is passed as a `void` pointer, any data type may be passed arrays, structs, - even C++ objects. This does mean that there are no type checks on the data that gets passed through. Additionally, because the key provider code gets called while performing an encode or decode operation - and not before or after, it is important not to change any fields in the `syncp_packet` structure, if the passback data contains pointers to or inside the structure.

```
    size_t key_provider(sym128_key_t *, _sp_key_id_t *, unsigned, void *);
```

```
int main()
{
    initialize_syncp();
    set_sym128_key_provider(key_provider);

    // Example buffer - assume it contains a syncp packet
    unsigned char packet_buffer[1024];
    size_t packet_size = 512;

    syncp_packet decode;
    unsigned char decoded_plaintext[64];

    decode_init(&packet_decode, packet_buffer, packet_size);
    set_passback_data("Hello World!");

    decode_packet(&packet_decode, decoded_plaintext, 64);

    return 0;
}

size_t key_provider(sym128_key_t *key, _sp_key_id_t *key_id, unsigned key_index, void *pb)
{
    char *str = (char *) pb; // str now contains "Hello World!"

    return 0; // Return an error - no keys exist in this example
}
```

Debug Logging

Debug logs are very helpful when building and testing an application. SyncP includes a very lightweight debug logging framework to aiding in debugging the library and application code. To start, compile the SyncP library with the `DEVELOP` flag defined (on most C compilers, this can be accomplished by adding a `-DDEVELOP` option to `CFLAGS`). The library will automatically log messages with the `WARNING`, `ERROR`, or `CRITICAL` status to `stderr`.

By including `syncp/log.h`, this behavior can be altered, and debug statements added into application code, using `log_print`, or `status_message`. Both are macros which will log a message, however `status_message` is able to be used in a return statement, and also includes the file and line numbers. `log_print` simply prints the line to the log. Both accept `printf`-style format strings and variable argument lists.

```
#include <syncp/log.h>
#include <stdio.h>

int main() {
    const char *log_filename = "application.log";
    FILE * logfile = fopen(log_filename, "a");

    set_log_level(LOG_LEVEL_INFO);
    set_log_location(logfile);

    log_print(LOG_LEVEL_INFO,
        "Application finished initializing logging to '%s'", log_filename);

    return status_message(0, LOG_LEVEL_WARNING, "Application returning");
}
```

This will yeild a log file with the following lines:

```
[INFO] Application finshed initializing logging to 'application.log'
```

```
[WARNING] main.c:13 | Application returning
```

Once application development has reached the production stage, simply omit the `DEVELOP` flag, and all `log_print` statements will be optimized away and `status_message` reduced to the first parameter. See Appendix A for more information on these macros.

Pluggable Cryptographic Modules

Pluggable cryptographic modules allows SyncP to be extended with support for other cryptographic libraries without needing to modify SyncP code.

Built-in Modules

Two cryptographic modules are built-in to SyncP: OpenSSL and TomCrypt. If SyncP is compiled with both, it is possible to dynamically switch between the two backend modules. By default, OpenSSL is enabled in this case.

```
#include <syncp/syncp.h>
#include <syncp/syncp_utils.h>
#include <syncp/log.h>

int main() {
    // OpenSSL is enabled by default
    initialize_syncp();

    // Get TomCrypt library
    const syncp_crypto_library *tc = get_crypto(LIBRARY_TOMCRYPT);

    // Swap out OpenSSL for TomCrypt
    if (replace_crypto(tc))
        return status_message(1, LOG_LEVEL_ERROR, "Failed to load in TomCrypt library.");

    return 0;
}
```

Custom Modules

SyncP supports loading custom external modules to replace the built-in backend modules. Through these modules, this allows SyncP to interact with hardware security devices, like HSMs or SHEs. There are a few differences in this configuration that are important to point out:

1. Key providers no longer are required to provide the actual key.

SyncP only acts to pass the key from the application to the crypto backend. Since the crypto backend is custom, it may implement key handling however is necessary. This is useful when the key is protected in hardware and is inaccessable from the software components. In these cases, the key buffer may be populated with up to 16 bytes of data, usually some sort of ASCII or binary identifier that the hardware module can identify.

Note: if key identifiers are provided instead of an actual key, the `key_id` buffer is **required** to be populated.

2. SyncP can be built without Tomcrypt or OpenSSL

Normally, SyncP requires that one or both of Tomcrypt and Openssl are compiled into libcryptofunctions. In the case where a custom cryptographic module is loaded, neither library needs to be compiled, reducing code size considerably.

Example Custom Module

This example implements the functions necessary for symmetric signing and verifying. Do note - none of this is valid cryptography - it's purely for illustration and to make the example compile/run. A full code listing is available in `src/examples/pluggable_crypto.c`, some of the code here is abbreviated.

First off, the standard includes for SyncP:

```
#include <stdlib.h>
#include <string.h>
#include <syncp/syncp.h>
#include <syncp/syncp_utils.h>
#include <syncp/v1/v1_syncp_utils.h>
#include <syncp/key_store.h>
#include <crypt/cryptofunctions.h>
```

Next, the functions required for the desired operations are defined. In this case for symmetric signing, only the `symmetric_sign` and `symmetric_verify` functions are required. The exact implementation of these functions don't serve much purpose for this example, as it it is not cryptographically correct, but is available in the full listing.

```
int plug_symmetric_sign(const unsigned char *msg, size_t msg_len, unsigned char *sig_buf,
    unsigned long sig_buf_len, const unsigned char *key, size_t key_len, mac mac_type);

int plug_symmetric_verify(const unsigned char *msg, size_t msg_len,
    const unsigned char *sig_buf, unsigned long sig_buf_len, const unsigned char *key,
    size_t key_len, mac mac_type);

static const crypto_library plug_lib =
{
    .symmetric_sign        = plug_symmetric_sign,
    .symmetric_verify      = plug_symmetric_verify,
};
```

Similar to previous examples, a key provider function is implemented for the cryptograpic modes that are being utilized.

```
size_t key_provider_k128(sym128_key_t *key, _sp_key_id_t *key_id, unsigned key_index, void *pb)
{
    // Ignore the passback data
    (void) pb;

    const size_t n_keys = 2;

    if (key_index >= n_keys)
        return 0;

    const unsigned char key_ids[2][8] =
    {
        { 0xbe, 0x45, 0xcb, 0x26, 0x05, 0xbf, 0x36, 0xbe, },
        { 0xfc, 0x2e, 0x2c, 0x73, 0x07, 0x2b, 0xfa, 0x2b, }
    };
```

Note here that instead of providing the key itself, an identifier is provided in its place, as well as the first 8 bytes of the corresponding sha256 hash as the `key_id`. Internally, SyncP will generate the `key_id` field based upon the sha256 has of the key - providing the `key_id` here disables implicit hashing.

```
    const char *keys[2] = {"syncp_key_1", "syncp_key_2"};

    strncpy((char *) key, keys[key_index], 16);
    memcpy(key_id, key_ids[key_index], sizeof(*key_id));
    return 16;
}

int main()
{
```

The application code remains largely unchanged, the only difference is a call to `initialize_syncp()` is replaced with a call to `set_crypto_lib()`. `initialize_syncp` is used to set up the backend cryptographic libraries, seeding random number generators, setting configuration options, etc. and as such, is not necessary when using a custom pluggable module.

```
        if (set_crypto_lib(&plug_lib))
            return 1;

        set_sym128_key_provider(key_provider_k128);

        unsigned char plaintext[13] = "Hello World!";

        syncp_packet   packet_encode;
        unsigned int   has_message_id      = 0;
        unsigned int   has_esn             = 1;
        unsigned int   has_key_id          = 1;
        unsigned int   cpu_destination     = 2;
        unsigned short service_type        = 25;
        unsigned short command_type        = 10;
        unsigned char  cryptotype          = CRYPTOTYPE_SYM_SIGN_128_HMAC;
        _sp_esn_t      esn                 = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
        unsigned char  key_id              = 1;

        if (encode_init(&packet_encode, has_message_id, has_esn, has_key_id, cpu_destination,
            service_type, command_type, cryptotype, plaintext, 12, &esn, key_id, 0, 0, 0, 0))
            return 1;

        size_t buf_size = get_encoded_size(&packet_encode);
        unsigned char *buffer = malloc(buf_size);

        if (encode_packet(&packet_encode, buffer, &buf_size))
            return 1;

        syncp_packet packet_decode;

        if (decode_init(&packet_decode, buffer, buf_size))
            return 1;

        if (packet_decode.version != 1)
            return 1;

        size_t decoded_ciphertext_size = packet_decode.packet.v1.plaintext_size;
        unsigned char *decoded = malloc(decoded_ciphertext_size);

        if (decode_packet(&packet_decode, decoded, decoded_ciphertext_size))
            return 1;

        free(buffer);
        free(decoded);
        return 0;
    }
```

Common Pitfalls

Buffer Too Small

SyncP runs internal checks on all buffers passed in to ensure that it does not corrupt memory. In some cases, a fixed buffer may not be large enough to hold a packet's encoded or decoded data, and so will fail. In general, it is recommended to verify in application code whether a buffer will be large enough for a particular operation. For example, an error will result when encoding the following:

```
unsigned short cryptotype = CRYPTOTYPE_ASYM_ENVELOPE_SIGN_RSA;
encode_init(&packet, ...);

unsigned char packet_buffer[256];
size_t buffer_size = 256;
if (encode_packet(&packet, packet_buffer, &buffer_size))
    log_print(LOG_LEVEL_ERROR, "Failed to encode packet");
```

This will result in a failure, as a 256 byte buffer is insufficent to store an encoded RSA enveloped packet. Instead, if the application is able, it is recommended to dynamically allocate a buffer. This can be accomplished with heap memory via the `*alloc` family, or on the stack with `alloca` and Variable Length Arrays. `get_encoded_size` will return the smallest size that still guarantees enough space, and will avoid the error. *Note: Microsoft compilers generally do not support VLA*:

```
size_t buf_size = get_encoded_size(&encode);
unsigned char packet_buffer[buf_size];

if (encode_packet(&packet, packet_buffer, &buffer_size))
    log_print(LOG_LEVEL_ERROR, "Failed to encode packet");
```

It is not always possible to use dynamic allocation, and in such cases, a call to `get_encoded_size` will allow an application to gracefully recover, as SyncP only returns that an error has occurred. In general, RSA-cryptotype messages are large compared to symmetric and EC cryptotype messages due to the size of the resulting signature and encrypted key.

Key ID Mismatch

Key IDs are a new concept in this revision of the protocol, and were added to aid in detecting rekeying issues. Essentially, the key ID is a sha256 hash of the key (or public key, if asymmetric), and is compared on the decoding side. If the key IDs do not match, SyncP will return a warning, and halt further processing of the packet.

```
[WARNING] Asym sign/verify key ID mismatch
```

If available, an application may look up precalculated sha256 hashes and write it to the `key_id` buffer to speed up calculations on embedded systems. The key ID comparison also helps serve as an early bail-out point when doing more expensive asymmetric calculations on embedded devices. This key ID check also serves to ensure development, testing, and verification environments are operating with the correct key material, as past versions without integrety checks or key checks may have simply returned garbage data.

If this warning is encountered, first ensure that matching keys are provided by matching keyslots. Next, ensure that the keyprovider for the key type is providing the correct key, and, if looking the key ID in a precomputed table, that both (a) the table entry is valid for the key, and (b) that the correct entry is being selected.

Upgrading from SyncP v1

Upgrading from the previous version of the SyncP library is relatively straightforward, as a majority of the functionality is supported. Before migration is preformed, however, it is important to note that this version of the SyncP library is not 100% backwards compatable with the previous version. Additionaly, error codes have changed, and SyncP v1 return codes are not able to be used when dealing with this new library.

Feature Compatability Table

SyncP supports most, but not all features of the previous protocol version.

v1 Feature	Support
No Security	Yes
Encryption Only	No
Integrity Only	Yes
Encryption + Integrity	Yes
High Bandwidth	Yes
Low Bandwidth	Decode Only

Example Upgrade

Here is some example application code from SyncP v1. Much in the same way as in previous examples, this code initializes, encodes, and decodes the same packet. Upgrading will be composed of three stages:

1. Setting up Key Providers
2. Converting struct setup to a function call
3. Simplifying the decode step

In this case, since everything is set at compile-type (i.e. the hardcoded key), there's not much to do in terms of simplifying the decode. In a real application, however, the key used will not necessarily be able to be determined at compile time, and the application would need to peek at the headers to determine which key to use.

```
int main()
{
    syncp_packet packet_encode, packet_decode;
    unsigned char packet_buffer[128];
    int status;

    memset(&packet_encode, 0, sizeof(packet_encode));
    memset(&packet_decode, 0, sizeof(packet_decode));
    memset(packet_buffer,  0, sizeof(packet_buffer));

    packet_encode.security_mode    = FLAG_ENCRYPTION_INTEGRITY; /* security mode */
    packet_encode.high_bandwidth   = 0;                         /* high bandwidth */
    packet_encode.response_req     = 0;                         /* response required */
    packet_encode.esn_req          = FLAG_ESN_REQ;              /* esn present */
    packet_encode.cpu              = 1;                         /* cpu */
    packet_encode.cpu_key          = 0;                         /* cpu_key */
    packet_encode.protocol_version = 0;                         /* protocol version */
    packet_encode.service_type     = 'a';                       /* message type */
    packet_encode.payload_size     = 4;                         /* payload size */
    memcpy(packet_encode.esn, "abcdefgh", ESN_LEN);             /* esn */
    memcpy(packet_encode.iv,  "halleymb", L_IV_LEN);            /* iv */

    /* Allocate and set payload */
    packet_encode.payload_alloc_size = packet_encode.payload_size;
    packet_encode.payload            = malloc(packet_encode.payload_size);
    memcpy(packet_encode.payload, "test", packet_encode.payload_size);

    unsigned char *enc_key = (unsigned char *) "sync_example_key";

    /* Encode Packet */
    unsigned packet_size = sizeof(packet_buffer);
    if (status = syncp_encode_packet(packet_buffer, &packet_size, &packet_encode, enc_key))
```

```
            return status;

        /* Decode Packet */
        if (status = syncp_decode_header(packet_buffer, packet_size, &packet_decode))
            return status;

        // Look up key required, and load into buffer
        unsigned char *dec_key = (unsigned char *) "sync_example_key";

        if (status = syncp_decode_packet(packet_buffer, packet_size, &packet_decode, dec_key)
            return status;

        return 0;
    }
```

Step one is to define a key provider. Since all keys in SyncP v1 are 128bit symmetric keys, we'll only need this one key provider.

```
    size_t key_provider_k128(sym128_key_t *key, _sp_key_id_t *key_id, int key_index, void *pb)
    {
        // Since this is a protocol v0 request, key_id will be null,
        // since key_id is a protocol v1 feature.
        (void) key_id;

        // Passback data is not used;
        (void) pb;

        // Hardcoded key
        unsigned char sym_key[16] = "sync_example_key";

        // Return 0 on error (requested key does not exist)
        if (key_index != 0)
            return 0;

        // Copy key into key buffer, if provided
        if (key)
            memcpy(key, sym_key, sizeof(sym_key));

        // Return number of bytes of the key
        return sizeof(sym_key);
    }
```

Next, packet initialization is switched to invoke `v0_encode_init`

```
    int main()
    {
        initialize_syncp();
        set_sym128_key_provider(key_provider_k128);

        syncp_packet packet_encode, packet_decode;
        unsigned char packet_buffer[128];

        memset(packet_buffer,  0, sizeof(packet_buffer));

        unsigned char response_required   = 0;
        unsigned char has_esn             = 1;
        unsigned char cpu_destination     = 1;
        unsigned char service_type        = 'a';
        unsigned char command_type        = 0;
        unsigned char cryptotype          = CRYPTOTYPE_V0_SYM_ENCRYPT_128_CCM;
        unsigned char key_index            = 0;

        _sp_esn_t esn = "abcdefgh";

        const char *plaintext = "test";
        unsigned long plaintext_length = strlen(plaintext);

        v0_encode_init(
            &packet_encode,
            response_required,
            cryptotype,
            has_esn,
            service_type,
            command_type,
            cpu_destination,
            key_index,
            (unsigned char *) plaintext,
            plaintext_length,
            &esn,
            0, 0, 0            /* Module + server message IDs, message status */
        );
```

Encoding the packet is much the same as before, excecpt the key is no longer needed to be provided.

```
        size_t packet_size = sizeof(packet_buffer);
        if (encode_packet(&packet_encode, packet_buffer, &packet_size))
            return 1;
```

Decoding similiarly gets simplified, as the key handling is performed by the key provider. SyncP will look up the key as necessary and call the key provider function defined (`key_provider_k128`).

```
        /* Decode Packet */
        unsigned char decoded_plaintext[16];
        if (decode_init(&packet_decode, packet_buffer, packet_size))
            return 1;

        if (decode_packet(&packet_decode, decoded_plaintext, 16))
            return 1;

        return 0;
    }
```

The SyncP Build System

Configure and Make

SyncP's build system is designed to be easy to use across as many platforms as possible, as well as easily automatable. Building the SyncP library follows the standard `./configure`, `make` sequence.

To build the SyncP library on a *nix platform:

```
$ ./configure -OT
$ make
```

This will result in the generation of the following library build artifacts:

- libcryptofunctions.a
- libsyncp.a

- libcryptofunctions.so
- libsyncp.so

As well as two test artifacts, `test-static.exe` and `test-dynamic.exe`. These will be covered in more detail in the next section. These artifacts are simply named with a `.exe` extension, and are valid executables on the build platform (ELF format on Linux).

Configure has the following options:

```
-T  Use Tomcrypt (From Source)
-O  Use OpenSSL (From Library)
-d  Use .dll instead of .so
-h  Shows this help menu

-n <arch>  Configure for nmake instead of make. Must specify either x86 or x64.
-s <suite> Use the provided test suite instead of the default unit-tests.
-F <flags> Add compilation flags
-L <flags> Add linking flags
```

There are two options for building on windows. Either (a) building with a POSIX compatability layer, such as CYGWIN or MSYS2, or with nmake. Building with a compatibility layer follows the same steps as building for a *nix platform, except with the `-d` flag to name the artifacts with a `.dll` extension.

If building with nmake, the `-n` flag must be specified along with the desired build architecture, either `x86` or `x64`. Note, that in all cases, the configure script is required to be run in a bash shell.

```
$ ./configure -OTdn x64
```

Then, in the Visual Studio command prompt, running `nmake` is sufficient to build the library and test executables.

```
> nmake
```

Common Build Flags

There are two build flags that will commonly be used with SyncP, especially if modifying and/or developing with the library.

- -DDEVELOP
- -DTEST_LARGE_OBJ

The first (`-DDEVELOP`) enables the debug logging features within SyncP, which are disabled by default as they do not conform to the MISRA C specification, due to the use of files - `fprintf()` in particular - as many embedded systems have no concept of files. This may, if desired for the particular application, be left enabled on a production system if logging is important. Because SyncP only returns that an error has occurred, and not the cause of the error, the debug logs are often the best way to trace issues to their source.

The second (`-DTEST_LARGE_OBJ`) enables the unit test library to generate large packets (up to 32 MiB) when when performing randomized library testing. This is disabled by default, as the buffers are heap-allocated, which may cause problems with memory monitoring tools, as SyncP does not use heap memory internally, except as required for OpenSSL operations.

Do note, that any C compiler flag may be included here to override the default compilation options. For example, `-g3` may be useful in conjunction to `-O0` to generate un-optomized assemblies with debug symbols for use in a debugger.

Thus, to configure the SyncP compilation with debug symbols:

```
$ ./configure -OTF "-DDEVELOP -g3 -O0"
```

The SyncP Unit Test Suite

SyncP has a robust suite of unit tests, designed to evaluate correctness across a wide range of functionality. Where possible, the test suite compares the packets generated to known valid packets. In most cases, however, this is not an option due to SyncP internally randomizing IV values. The test suite compensates for this by cross-encoding and cross-decoding packets between OpenSSL and the built-in version of TomCrypt, and measures validity by the ability of each library to decode packets generated by the other.

To take advantage of these cross checks, both OpenSSL and TomCrypt are required to be configured (the `-O` and `-T` flags for the configure script). If built with debug logging (`-DDEVELOP`), running the test executables will generate a visual indication of the test results:

```
$ ./test-static.exe
[INFO] Test 'v0_syncp_base encoding' passed
[INFO] Test 'v0_syncp_base decode encoded message' passed
[INFO] Test 'v0_syncp_cmac128_sign openssl encode openssl decode' passed
[INFO] Test 'v0_syncp_cmac128_sign tomcrypt encode tomcrypt decode' passed
[INFO] Test 'v0_syncp_cmac128_sign openssl encode tomcrypt decode' passed
[INFO] Test 'v0_syncp_cmac128_sign tomcrypt encode openssl decode' passed
...
```

The `test-static.exe` executable is statically-linked with the SyncP library, and the `test-dynamic.exe` executable is dynamically linked with the SyncP library. This allows troubleshooting dynamic linking issues with known-good assemblies.