RSA Signature Algorithm User Guide

Version 1.50 BETA

For use with RSA module versions 2.05 and above

Table of Contents

1. System Overview 3

1.1. Introduction 4

1.2. Feature Check 7

1.3. Packages and Documents 8

1.4. Change History 9

2. Source File List 10

3. Configuration Options 11

4. Application Programming Interface 12

4.1. Functions 12

rsa_init_fn 13

rsa_raw_init_fn 14

rsa_register_tests 15

4.2. Key Lengths 16

4.3. Padding Types 16

4.4. OID Numbers 16

4.5. OID Lengths 16

4.6. Error Codes 17

5. Integration 18

5.1. OS Abstraction Layer 18

5.2. PSP Porting 19

6. Version 20

1. System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

- Introduction – describes the main elements of the module.
- Feature Check – summarizes the main features of the module as bullet points.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

1.1. Introduction

This guide is for those who want to implement encryption using the Rivest, Shamir, and Adelman (RSA) signature algorithm. This module is part of the CryptoCore™ Base security suite.

Overview

The RSA encryption algorithm is used as a signing and key exchange mechanism.

The RSA private key is used to encrypt a shared secret value and send it to the peer. The public key is used to decrypt received values. Therefore RSA is an asymmetric algorithm.

RSA keys can be 1024, 2048, or 4096 bits (128, 256, or 512 bytes, respectively). Numerous tools are available for generating RSA keys.

Note:

- RSA was very popular for certificate validation but is slowly being replaced by ECDSA which is more computationally efficient and secure. ECDSA is available as part of HCC's ECC module.
- RSA is not normally used for encryption because it is too computationally intense.

RSA encryption/decryption can use both private RSA keys (X.509 standard) and public keys (X.509 certificate subject public key information).

Data is signed by RSA by calculating the hash value of given data and then encrypting it. In some cases the hash value is preceded by an OID number that identifies the used hash value. (The X.509 signing standard adds the OID number before the hash. In X.509 you encrypt the DER-encoded OID and the hash value.)

The algorithm is not stateful but multiple instances can be configured.

The maximum public key length can also be configured.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to encrypt input data.

p_in[] points to the data to be signed. The length of the data (*in_len*) does not need to be aligned. *in_len* must be shorter than or equal to the key size used - 11 bytes, as follows:

- For 1024 bit, 117 bytes.
- For 2048 bit, 245 bytes.
- For 4096 bit, 501 bytes.

The input data can have 1 to 117 bytes when we use a 1024 bit key (two bytes are needed to add proper padding).

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
ecd_init_vect_size	uint16_t	0 - random padding. 1 - padding filled with 0xFF. 2 - padding filled with 0x00.
p_ecd_key	void *	A pointer to the buffer storing the DER-encoded key.
ecd_key_size	uint16_t	The length of the key in bytes. This is the size of the RSA public/private key in the format standardized by X.509. This is not 128, 256, or 512 but the size of the DER-encoded public/private key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the encrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the size of the key used (128, 256, or 512 bytes).

enc_driver_decrypt()

The EEM function **enc_driver_decrypt()** is used to check the signature of given data.

p_in[] points to the data to be decrypted. This must be not greater than the key resolution (128, 256, or 512).

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_key	void *	A pointer to the buffer storing the RSA key.
ecd_key_size	uint16_t	The length of the public key in bytes.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_decrypt()** is the decrypted data, *p_out[]*.

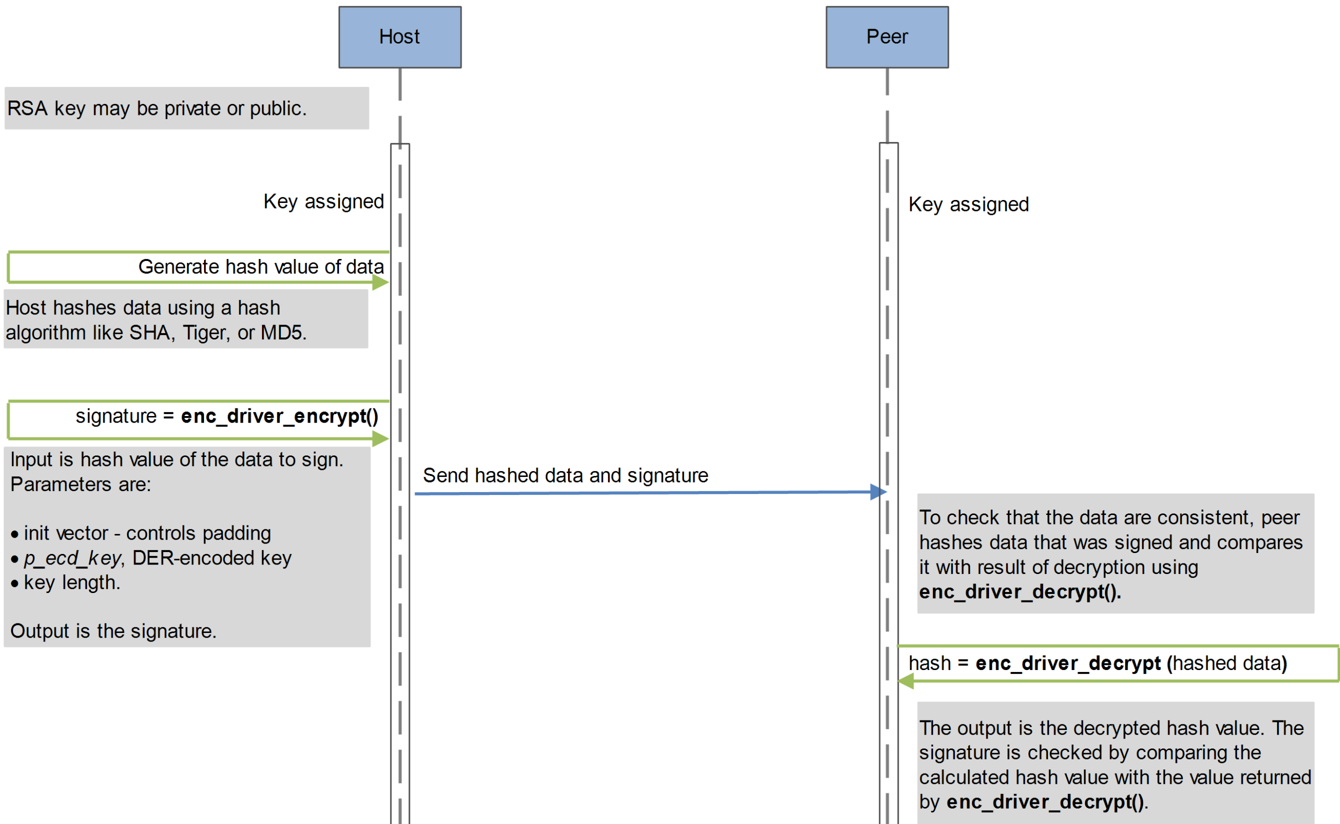
The output length, *p_out_len*, must be set to the length of the decrypted data. This must not be greater than the size of the key (128, 256, or 512 bytes).

The algorithm is not stateful but multiple instances can be configured. The maximum public key length can also be configured.

Copyright HCC Embedded 2021

Sequence Diagram

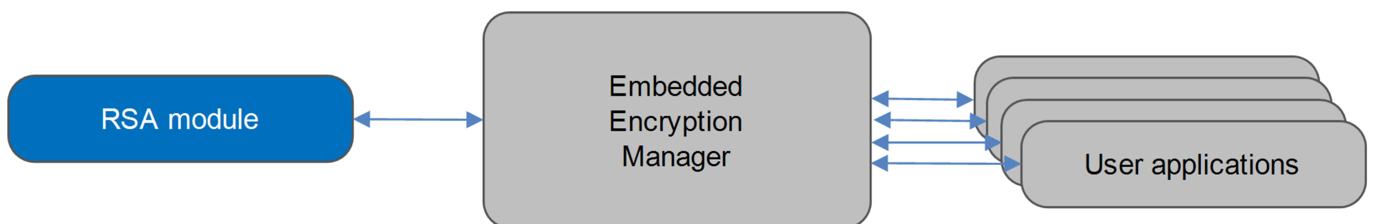
The following sequence diagram shows the process:



Using the Module

You register the RSA module with HCC's Embedded Encryption Manager (EEM), making it usable by other applications (for example, HCC's TLS/DTLS) through a standard interface. The EEM is the core component of HCC's encryption system.

The system structure is shown below:



A complete test suite is included for validating all of the HCC algorithms.

Note:

- Although every attempt has been made to simplify the system's use, to get the best results you must understand clearly the requirements of the systems you design.
- HCC Embedded offers hardware and firmware development consultancy to help you implement your system; contact sales@hcc-embedded.com.

Copyright HCC Embedded 2021

1.2. Feature Check

The main features of the RSA module are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Conforms to the HCC Coding Standard including full MISRA compliance.
- Designed for integration with both RTOS and non-RTOS based systems.
- Conforms to the HCC Embedded Encryption Manager (EEM) standard and is compatible with the EEM.
- Supports a maximum key length of 4096 bits (512 bytes).
- Supports the RSA CRT algorithm (see RFC 2437 section 5.2.1).
- Integral test suite gives complete logical coverage test of the algorithm.

1.3. Packages and Documents

Packages

The table below lists the packages that you need in order to use this module.

Package	Description
hcc_base_docs	This contains the two guides that will help you get started.
enc_base	The EEM base package.
enc_rsa	The RSA package described in this document.

Documents

For an overview of HCC verifiable embedded network encryption, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC Embedded Encryption Manager User Guide

This document describes the EEM.

HCC Encryption Test Suite User Guide

This document describes how to run tests to validate the algorithm.

HCC RSA Signature Algorithm User Guide

This is this document.

1.4. Change History

This section describes past changes to this manual.

- To download this manual or a PDF describing an earlier software version, see Encryption PDFs.
- For the history of changes made to the package code itself, see History: enc_rsa.

The current version of this manual is 1.50 BETA. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
1.50B	2021-01-14	2.05	Added Raw configuration options. Added **rsa_raw_init_fn()** function. Added RSA_SHA384_ALG_OID and *OID Lengths* section.
1.40B	2018-02-22	2.02	Extended *Introduction*, added new test options and function.
1.30B	2017-06-15	2.01	New *Change History* format.
1.20B	2017-01-10	2.00	Added *Padding Types*.
1.10B	2016-03-18	1.04	Added Change History.
1.00B	2015-02-11	1.03	First online version.

2. Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

Note: Do not modify any files except the configuration file.

API Header File

The file **src/api/api_enc_sw_rsa.h** is the only file that should be included by an application using this module. It defines the Application Programming Interface (API) functions.

Configuration File

The file **src/config/config_enc_sw_rsa.h** contains the configurable system parameters. Configure these as required. This is the only file in the module that you should modify.

System File

The file **src/enc/software/rsa/rsa.c** is the source code file. **This file should only be modified by HCC**.

Test File

The file **src/enc/test/test_rsa.c** contains the test registration source code. **This file should only be modified by HCC**.

Version File

The file **src/version/ver_enc_sw_rsa.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

3. Configuration Options

Set the system configuration options in the file **src/config/config_enc_sw_rsa.h**.

RSA_INSTANCE_NR

The maximum number of RSA driver instances. The default is 3.

RSA_RAW_INSTANCE_NR

The maximum number of RSA PSS instances. The default is 2.

RSA_MAX_PUB_KEY_LEN

The maximum RSA public key length in bytes. This must be less than or equal to the maximum length of a supported big number value (SBN_BUF_LEN in the base package's **config_enc.h** file). The default is 256.

Note: The RSA strength is usually quoted in bits but this option uses bytes. The default of 256 bytes is 2048 bits.

RSA_TEST_ENABLE

Set this to 1 to enable the RSA test suite. The default is 0.

RSA_TEST_RSA_INITFN

The RSA encryption driver initialization function. The default is *&rsa_init_fn*; redefine this if you want to use another set of drivers for a compatibility check.

RSA_RAW_TEST_ENABLE

Set this to 1 to enable the RSA Raw test suite. The default is 0.

RSA_RAW_TEST_RSA_INITFN

The RSA Raw encryption driver initialization function. The default is *&rsa_raw_init_fn*; redefine this if you want to use another set of drivers for a compatibility check.

Copyright HCC Embedded 2021

4. Application Programming Interface

This section describes the single Application Programming Interface (API) function, the key lengths, the padding types, and the error codes.

4.1. Functions

Call the initialization function from the EEM to register the algorithm with it. Call the test function to register the RSA tests with the EEM test module.

The functions are the following:

Function	Description
rsa_init_fn()	Called from the EEM, this registers the RSA algorithm with it.
rsa_raw_init_fn()	Called from the EEM, this registers the RSA Raw algorithm with it.
rsa_register_tests()	Registers the RSA tests with the EEM test module.

rsa_init_fn

Call this initialization function from the EEM to register the algorithm with it.

This forwards the *t_enc_driver_fn* structure containing RSA functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*.

Format

```
t_enc_ret rsa_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing RSA functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

rsa_raw_init_fn

Call this initialization function from the EEM to register the algorithm with it.

This forwards the *t_enc_driver_fn* structure containing Raw functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*.

Format

```
t_enc_ret rsa_raw_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing RSA functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

rsa_register_tests

Call this function to register the RSA tests with the EEM test module.

Once you have registered the tests, you can execute the test suite as directed in the *HCC Encryption Test Suite User Guide*.

Note: The RSA_TEST_ENABLE configuration option must be set to 1 to enable this function.

Format

```
t_enc_ret rsa_register_tests ( void )
```

Arguments

None.

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
Else	See Error Codes.

4.2. Key Lengths

The minimum key length is defined in the file **src/api/api_enc_sw_rsa.h**.

The maximum public key length in bytes, RSA_MAX_PUB_KEY_LEN, is a configuration option. Its default is 256.

Name	Value	Description
RSA_MIN_PUB_KEY_LEN	64	The minimum public key length in bytes.

4.3. Padding Types

The padding types are defined in the file **src/api/api_enc_sw_rsa.h**.

Name	Value	Description
RSA_PADDING_RANDOM	0	The signature padding is a pseudorandom value.
RSA_PADDING_SET	1	The signature padding is filled with 0xFF values.
RSA_PADDING_CLEAR	2	The signature padding is filled with 0x00 values.

4.4. OID Numbers

The OID numbers for different RSA signature methods are as follows:

Name	Value	Description
RSA_SHA_ALG_OID	{0x2AU, 0x86U, 0x48U, 0x86U, 0xF7U, 0x0DU, 0x01U, 0x01U, 0x05U}	RSA SHA.
RSA_SHA256_ALG_OID	{0x2AU, 0x86U, 0x48U, 0x86U, 0xF7U, 0x0DU, 0x01U, 0x01U, 0x0BU}	RSA SHA-256.
RSA_SHA384_ALG_OID	{0x2AU, 0x86U, 0x48U, 0x86U, 0xF7U, 0x0DU, 0x01U, 0x01U, 0x0CU}	RSA SHA-384.

4.5. OID Lengths

The length of the RSA signature OIDs are as follows:

Name	Value	Description
TLS_RSA_SHA_SIGN_OID_LEN	9	Length of the RSA-SHA OID.
TLS_RSA_SHA256_SIGN_OID_LEN	9	Length of the RSA-SHA256 OID.
TLS_RSA_SHA384_SIGN_OID_LEN	9	Length of the RSA-SHA384 OID.

4.6. Error Codes

The table below lists the error codes that may be generated by the API call.

Error code	Value	Meaning
ENC_SUCCESS	0	Successful execution.
ENC_INVALID_ERR	1	The module has already been initialized.

5. Integration

The RSA module is designed to be as open and as portable as possible. No assumptions are made about the functionality, the behavior, or even the existence, of the underlying operating system. For the system to work at its best, perform the porting outlined below. This is a straightforward task for an experienced engineer.

5.1. OS Abstraction Layer

The module uses the OS Abstraction Layer (OAL) that allows it to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The module uses the following OAL components:

OAL Resource	Number Required
Tasks	0
Mutexes	1 per algorithm used
Events	0

5.2. PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer. For full details of these elements, see the *HCC Base Platform Support Package User Guide*.

The module makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.
PSP_WR_8BITARRAY_OFFSET()	psp_base	psp_array	Writes the offset in an 8 bit array.

The module uses the following big number arithmetic functions from the EEM's Big Number Arithmetic API. These are described in the the *HCC Embedded Encryption Manager User Guide*.

Note: To improve performance, you can replace these functions with optimized or hardware-supported versions.

Function	Description
bn_add()	Adds two big numbers.
bn_assign_be_buf()	Assigns a little-endian buffer to a big number, based on a big-endian buffer.
bn_compare()	Compares two large numbers.
bn_get_be_buf()	Exports a big number to a big-endian buffer.
bn_get_power_modulo()	Calculates *p_a* raised to the power of *p_e*, modulo *p_m*, and stores the result in *p_r*.
bn_modular_multiplication()	Counts *a*b* mod *modulo* using the Montgomery algorithm.
bn_modulo()	Calculates the remainder of *p_a* divided by *p_mod*.
bn_mul()	Multiplies one big number by another.
bn_subtract()	Subtracts one big number from another.

Copyright HCC Embedded 2021

6. Version

Version 1.50 BETA

For use with RSA module versions 2.05 and above