Linux Kernel Crypto API manual

Release

The kernel development community

August 18, 2018

CONTENTS

Author Stephan Mueller

Author Marek Vasut

This documentation outlines the Linux kernel crypto API with its concepts, details about developing cipher implementations, employment of the API for cryptographic use cases, as well as programming examples.

Table of contents

CHAPTER

ONE

KERNEL CRYPTO API INTERFACE SPECIFICATION

Introduction

The kernel crypto API offers a rich set of cryptographic ciphers as well as other data transformation mechanisms and methods to invoke these. This document contains a description of the API and provides example code.

To understand and properly use the kernel crypto API a brief explanation of its structure is given. Based on the architecture, the API can be separated into different components. Following the architecture specification, hints to developers of ciphers are provided. Pointers to the API function call documentation are given at the end.

The kernel crypto API refers to all algorithms as "transformations". Therefore, a cipher handle variable usually has the name "tfm". Besides cryptographic operations, the kernel crypto API also knows compression transformations and handles them the same way as ciphers.

The kernel crypto API serves the following entity types:

- consumers requesting cryptographic services
- data transformation implementations (typically ciphers) that can be called by consumers using the kernel crypto API

This specification is intended for consumers of the kernel crypto API as well as for developers implementing ciphers. This API specification, however, does not discuss all API calls available to data transformation implementations (i.e. implementations of ciphers and other transformations (such as CRC or even compression algorithms) that can register with the kernel crypto API).

Note: The terms "transformation" and cipher algorithm are used interchangeably.

Terminology

The transformation implementation is an actual code or interface to hardware which implements a certain transformation with precisely defined behavior.

The transformation object (TFM) is an instance of a transformation implementation. There can be multiple transformation objects associated with a single transformation implementation. Each of those transformation objects is held by a crypto API consumer or another transformation. Transformation object is allocated when a crypto API consumer requests a transformation implementation. The consumer is then provided with a structure, which contains a transformation object (TFM).

The structure that contains transformation objects may also be referred to as a "cipher handle". Such a cipher handle is always subject to the following phases that are reflected in the API calls applicable to such a cipher handle:

1. Initialization of a cipher handle.
2. Execution of all intended cipher operations applicable for the handle where the cipher handle must be furnished to every API call.

3. Destruction of a cipher handle.

When using the initialization API calls, a cipher handle is created and returned to the consumer. Therefore, please refer to all initialization API calls that refer to the data structure type a consumer is expected to receive and subsequently to use. The initialization API calls have all the same naming conventions of crypto_alloc*.

The transformation context is private data associated with the transformation object.

CHAPTER

TWO

KERNEL CRYPTO API ARCHITECTURE

Cipher algorithm types

The kernel crypto API provides different API calls for the following cipher types:

- Symmetric ciphers
- AEAD ciphers
- Message digest, including keyed message digest
- Random number generation
- User space interface

Ciphers And Templates

The kernel crypto API provides implementations of single block ciphers and message digests. In addition, the kernel crypto API provides numerous "templates" that can be used in conjunction with the single block ciphers and message digests. Templates include all types of block chaining mode, the HMAC mechanism, etc.

Single block ciphers and message digests can either be directly used by a caller or invoked together with a template to form multi-block ciphers or keyed message digests.

A single block cipher may even be called with multiple templates. However, templates cannot be used without a single cipher.

See /proc/crypto and search for "name". For example:

- aes
- ecb(aes)
- cmac(aes)
- ccm(aes)
- rfc4106(gcm(aes))
- sha1
- hmac(sha1)
- authenc(hmac(sha1),cbc(aes))

In these examples, "aes" and "sha1" are the ciphers and all others are the templates.

Synchronous And Asynchronous Operation

The kernel crypto API provides synchronous and asynchronous API operations.

When using the synchronous API operation, the caller invokes a cipher operation which is performed synchronously by the kernel crypto API. That means, the caller waits until the cipher operation completes. Therefore, the kernel crypto API calls work like regular function calls. For synchronous operation, the set of API calls is small and conceptually similar to any other crypto library.

Asynchronous operation is provided by the kernel crypto API which implies that the invocation of a cipher operation will complete almost instantly. That invocation triggers the cipher operation but it does not signal its completion. Before invoking a cipher operation, the caller must provide a callback function the kernel crypto API can invoke to signal the completion of the cipher operation. Furthermore, the caller must ensure it can handle such asynchronous events by applying appropriate locking around its data. The kernel crypto API does not perform any special serialization operation to protect the caller's data integrity.

Crypto API Cipher References And Priority

A cipher is referenced by the caller with a string. That string has the following semantics:

```
template(single block cipher)
```

where "template" and "single block cipher" is the aforementioned template and single block cipher, respectively. If applicable, additional templates may enclose other templates, such as

```
template1(template2(single block cipher)))
```

The kernel crypto API may provide multiple implementations of a template or a single block cipher. For example, AES on newer Intel hardware has the following implementations: AES-NI, assembler implementation, or straight C. Now, when using the string "aes" with the kernel crypto API, which cipher implementation is used? The answer to that question is the priority number assigned to each cipher implementation by the kernel crypto API. When a caller uses the string to refer to a cipher during initialization of a cipher handle, the kernel crypto API looks up all implementations providing an implementation with that name and selects the implementation with the highest priority.

Now, a caller may have the need to refer to a specific cipher implementation and thus does not want to rely on the priority-based selection. To accommodate this scenario, the kernel crypto API allows the cipher implementation to register a unique name in addition to common names. When using that unique name, a caller is therefore always sure to refer to the intended cipher implementation.

The list of available ciphers is given in /proc/crypto. However, that list does not specify all possible permutations of templates and ciphers. Each block listed in /proc/crypto may contain the following information – if one of the components listed as follows are not applicable to a cipher, it is not displayed:

- name: the generic name of the cipher that is subject to the priority-based selection – this name can be used by the cipher allocation API calls (all names listed above are examples for such generic names)
- driver: the unique name of the cipher – this name can be used by the cipher allocation API calls
- module: the kernel module providing the cipher implementation (or "kernel" for statically linked ciphers)
- priority: the priority value of the cipher implementation
- refcnt: the reference count of the respective cipher (i.e. the number of current consumers of this cipher)
- selftest: specification whether the self test for the cipher passed
- type:
  - skcipher for symmetric key ciphers

  - cipher for single block ciphers that may be used with an additional template
  - shash for synchronous message digest
  - ahash for asynchronous message digest
  - aead for AEAD cipher type
  - compression for compression type transformations
  - rng for random number generator
  - givcipher for cipher with associated IV generator (see the geniv entry below for the specification of the IV generator type used by the cipher implementation)
  - kpp for a Key-agreement Protocol Primitive (KPP) cipher such as an ECDH or DH implementation
- blocksize: blocksize of cipher in bytes
- keysize: key size in bytes
- ivsize: IV size in bytes
- seedsize: required size of seed data for random number generator
- digestsize: output size of the message digest
- geniv: IV generation type:
  - eseqiv for encrypted sequence number based IV generation
  - seqiv for sequence number based IV generation
  - chainiv for chain iv generation
  - <builtin> is a marker that the cipher implements IV generation and handling as it is specific to the given cipher

Key Sizes

When allocating a cipher handle, the caller only specifies the cipher type. Symmetric ciphers, however, typically support multiple key sizes (e.g. AES-128 vs. AES-192 vs. AES-256). These key sizes are determined with the length of the provided key. Thus, the kernel crypto API does not provide a separate way to select the particular symmetric cipher key size.

Cipher Allocation Type And Masks

The different cipher handle allocation functions allow the specification of a type and mask flag. Both parameters have the following meaning (and are therefore not covered in the subsequent sections).

The type flag specifies the type of the cipher algorithm. The caller usually provides a 0 when the caller wants the default handling. Otherwise, the caller may provide the following selections which match the aforementioned cipher types:

- CRYPTO_ALG_TYPE_CIPHER Single block cipher
- CRYPTO_ALG_TYPE_COMPRESS Compression
- CRYPTO_ALG_TYPE_AEAD Authenticated Encryption with Associated Data (MAC)
- CRYPTO_ALG_TYPE_BLKCIPHER Synchronous multi-block cipher
- CRYPTO_ALG_TYPE_ABLKCIPHER Asynchronous multi-block cipher
- CRYPTO_ALG_TYPE_GIVCIPHER Asynchronous multi-block cipher packed together with an IV generator (see geniv field in the /proc/crypto listing for the known IV generators)

- CRYPTO_ALG_TYPE_KPP Key-agreement Protocol Primitive (KPP) such as an ECDH or DH implementation
- CRYPTO_ALG_TYPE_DIGEST Raw message digest
- CRYPTO_ALG_TYPE_HASH Alias for CRYPTO_ALG_TYPE_DIGEST
- CRYPTO_ALG_TYPE_SHASH Synchronous multi-block hash
- CRYPTO_ALG_TYPE_AHASH Asynchronous multi-block hash
- CRYPTO_ALG_TYPE_RNG Random Number Generation
- CRYPTO_ALG_TYPE_AKCIPHER Asymmetric cipher
- CRYPTO_ALG_TYPE_PCOMPRESS Enhanced version of CRYPTO_ALG_TYPE_COMPRESS allowing for segmented compression / decompression instead of performing the operation on one segment only. CRYPTO_ALG_TYPE_PCOMPRESS is intended to replace CRYPTO_ALG_TYPE_COMPRESS once existing consumers are converted.

The mask flag restricts the type of cipher. The only allowed flag is CRYPTO_ALG_ASYNC to restrict the cipher lookup function to asynchronous ciphers. Usually, a caller provides a 0 for the mask flag.

When the caller provides a mask and type specification, the caller limits the search the kernel crypto API can perform for a suitable cipher implementation for the given cipher name. That means, even when a caller uses a cipher name that exists during its initialization call, the kernel crypto API may not select it due to the used type and mask field.

Internal Structure of Kernel Crypto API

The kernel crypto API has an internal structure where a cipher implementation may use many layers and indirections. This section shall help to clarify how the kernel crypto API uses various components to implement the complete cipher.

The following subsections explain the internal structure based on existing cipher implementations. The first section addresses the most complex scenario where all other scenarios form a logical subset.

Generic AEAD Cipher Structure

The following ASCII art decomposes the kernel crypto API layers when using the AEAD cipher with the automated IV generation. The shown example is used by the IPSEC layer.

For other use cases of AEAD ciphers, the ASCII art applies as well, but the caller may not use the AEAD cipher with a separate IV generator. In this case, the caller must generate the IV.

The depicted example decomposes the AEAD cipher of GCM(AES) based on the generic C implementations (gcm.c, aes-generic.c, ctr.c, ghash-generic.c, seqiv.c). The generic implementation serves as an example showing the complete logic of the kernel crypto API.

It is possible that some streamlined cipher implementations (like AES-NI) provide implementations merging aspects which in the view of the kernel crypto API cannot be decomposed into layers any more. In case of the AES-NI implementation, the CTR mode, the GHASH implementation and the AES cipher are all merged into one cipher implementation registered with the kernel crypto API. In this case, the concept described by the following ASCII art applies too. However, the decomposition of GCM into the individual sub-components by the kernel crypto API is not done any more.

Each block in the following ASCII art is an independent cipher instance obtained from the kernel crypto API. Each block is accessed by the caller or by other blocks using the API functions defined by the kernel crypto API for the cipher implementation type.

The blocks below indicate the cipher type as well as the specific logic implemented in the cipher.

The ASCII art picture also indicates the call structure, i.e. who calls which component. The arrows point to the invoked block where the caller uses the API applicable to the cipher type specified for the block.

```
kernel crypto API                                |   IPSEC Layer
                                                 |
+-----------+                                    |
|           |           (1)
|   aead    | <-----------------------------------  esp_output
|  (seqiv)  | ---+
+-----------+    |
                 | (2)
+-----------+    |
|           | <--+                (2)
|   aead    | <-----------------------------------  esp_input
|   (gcm)   | ------------+
+-----------+             |
      | (3)               | (5)
      v                   v
+-----------+       +-----------+
|           |       |           |
|  skcipher |       |   ahash   |
|   (ctr)   | ---+  |  (ghash)  |
+-----------+    |  +-----------+
                 |
+-----------+    | (4)
|           | <--+
|   cipher  |
|   (aes)   |
+-----------+
```

The following call sequence is applicable when the IPSEC layer triggers an encryption operation with the esp_output function. During configuration, the administrator set up the use of rfc4106(gcm(aes)) as the cipher for ESP. The following call sequence is now depicted in the ASCII art above:

1. esp_output() invokes crypto_aead_encrypt() to trigger an encryption operation of the AEAD cipher with IV generator.

   In case of GCM, the SEQIV implementation is registered as GIVCIPHER in crypto_rfc4106_alloc().

   The SEQIV performs its operation to generate an IV where the core function is seqiv_geniv().

2. Now, SEQIV uses the AEAD API function calls to invoke the associated AEAD cipher. In our case, during the instantiation of SEQIV, the cipher handle for GCM is provided to SEQIV. This means that SEQIV invokes AEAD cipher operations with the GCM cipher handle.

   During instantiation of the GCM handle, the CTR(AES) and GHASH ciphers are instantiated. The cipher handles for CTR(AES) and GHASH are retained for later use.

   The GCM implementation is responsible to invoke the CTR mode AES and the GHASH cipher in the right manner to implement the GCM specification.

3. The GCM AEAD cipher type implementation now invokes the SKCIPHER API with the instantiated CTR(AES) cipher handle.

   During instantiation of the CTR(AES) cipher, the CIPHER type implementation of AES is instantiated. The cipher handle for AES is retained.

   That means that the SKCIPHER implementation of CTR(AES) only implements the CTR block chaining mode. After performing the block chaining operation, the CIPHER implementation of AES is invoked.

4. The SKCIPHER of CTR(AES) now invokes the CIPHER API with the AES cipher handle to encrypt one block.

5. The GCM AEAD implementation also invokes the GHASH cipher implementation via the AHASH API.

When the IPSEC layer triggers the esp_input() function, the same call sequence is followed with the only difference that the operation starts with step (2).

Generic Block Cipher Structure

Generic block ciphers follow the same concept as depicted with the ASCII art picture above.

For example, CBC(AES) is implemented with cbc.c, and aes-generic.c. The ASCII art picture above applies as well with the difference that only step (4) is used and the SKCIPHER block chaining mode is CBC.

Generic Keyed Message Digest Structure

Keyed message digest implementations again follow the same concept as depicted in the ASCII art picture above.

For example, HMAC(SHA256) is implemented with hmac.c and sha256_generic.c. The following ASCII art illustrates the implementation:

```
kernel crypto API            |       Caller
                             |
+-----------+         (1)    |
|           | <------------------  some_function
|   ahash   |
|   (hmac)  | ---+
+-----------+    |
                 | (2)
+-----------+    |
|           | <--+
|   shash   |
|  (sha256) |
+-----------+
```

The following call sequence is applicable when a caller triggers an HMAC operation:

1. The AHASH API functions are invoked by the caller. The HMAC implementation performs its operation as needed.

   During initialization of the HMAC cipher, the SHASH cipher type of SHA256 is instantiated. The cipher handle for the SHA256 instance is retained.

   At one time, the HMAC implementation requires a SHA256 operation where the SHA256 cipher handle is used.

2. The HMAC instance now invokes the SHASH API with the SHA256 cipher handle to calculate the message digest.

CHAPTER

THREE

DEVELOPING CIPHER ALGORITHMS

Registering And Unregistering Transformation

There are three distinct types of registration functions in the Crypto API. One is used to register a generic cryptographic transformation, while the other two are specific to HASH transformations and COMPRESSion. We will discuss the latter two in a separate chapter, here we will only look at the generic ones.

Before discussing the register functions, the data structure to be filled with each, struct crypto_alg, must be considered – see below for a description of this data structure.

The generic registration functions can be found in include/linux/crypto.h and their definition can be seen below. The former function registers a single transformation, while the latter works on an array of transformation descriptions. The latter is useful when registering transformations in bulk, for example when a driver implements multiple transformations.

```
int crypto_register_alg(struct crypto_alg *alg);
int crypto_register_algs(struct crypto_alg *algs, int count);
```

The counterparts to those functions are listed below.

```
int crypto_unregister_alg(struct crypto_alg *alg);
int crypto_unregister_algs(struct crypto_alg *algs, int count);
```

Notice that both registration and unregistration functions do return a value, so make sure to handle errors. A return code of zero implies success. Any return code < 0 implies an error.

The bulk registration/unregistration functions register/unregister each transformation in the given array of length count. They handle errors as follows:

- crypto_register_algs() succeeds if and only if it successfully registers all the given transformations. If an error occurs partway through, then it rolls back successful registrations before returning the error code. Note that if a driver needs to handle registration errors for individual transformations, then it will need to use the non-bulk function crypto_register_alg() instead.
- crypto_unregister_algs() tries to unregister all the given transformations, continuing on error. It logs errors and always returns zero.

Single-Block Symmetric Ciphers [CIPHER]

Example of transformations: aes, arc4, ...

This section describes the simplest of all transformation implementations, that being the CIPHER type used for symmetric ciphers. The CIPHER type is used for transformations which operate on exactly one block at a time and there are no dependencies between blocks at all.

Registration specifics

The registration of [CIPHER] algorithm is specific in that struct crypto_alg field .cra_type is empty. The .cra_u.cipher has to be filled in with proper callbacks to implement this transformation.

See struct cipher_alg below.

Cipher Definition With struct cipher_alg

Struct cipher_alg defines a single block cipher.

Here are schematics of how these functions are called when operated from other part of the kernel. Note that the .cia_setkey() call might happen before or after any of these schematics happen, but must not happen during any of these are in-flight.

```
KEY ---.    PLAINTEXT ---.
       v                 v
 .cia_setkey() -> .cia_encrypt()
                         |
                         '-----> CIPHERTEXT
```

Please note that a pattern where .cia_setkey() is called multiple times is also valid:

```
KEY1 --.    PLAINTEXT1 --.         KEY2 --.    PLAINTEXT2 --.
       v                 v                v                 v
 .cia_setkey() -> .cia_encrypt() -> .cia_setkey() -> .cia_encrypt()
                         |                                  |
                         '---> CIPHERTEXT1                  '---> CIPHERTEXT2
```

Multi-Block Ciphers

Example of transformations: cbc(aes), ecb(arc4), …

This section describes the multi-block cipher transformation implementations. The multi-block ciphers are used for transformations which operate on scatterlists of data supplied to the transformation functions. They output the result into a scatterlist of data as well.

Registration Specifics

The registration of multi-block cipher algorithms is one of the most standard procedures throughout the crypto API.

Note, if a cipher implementation requires a proper alignment of data, the caller should use the functions of crypto_skcipher_alignmask() to identify a memory alignment mask. The kernel crypto API is able to process requests that are unaligned. This implies, however, additional overhead as the kernel crypto API needs to perform the realignment of the data which may imply moving of data.

Cipher Definition With struct blkcipher_alg and ablkcipher_alg

Struct blkcipher_alg defines a synchronous block cipher whereas struct ablkcipher_alg defines an asynchronous block cipher.

Please refer to the single block cipher description for schematics of the block cipher usage.

Specifics Of Asynchronous Multi-Block Cipher

There are a couple of specifics to the asynchronous interface.

First of all, some of the drivers will want to use the Generic ScatterWalk in case the hardware needs to be fed separate chunks of the scatterlist which contains the plaintext and will contain the ciphertext. Please refer to the ScatterWalk interface offered by the Linux kernel scatter / gather list implementation.

Hashing [HASH]

Example of transformations: crc32, md5, sha1, sha256,...

Registering And Unregistering The Transformation

There are multiple ways to register a HASH transformation, depending on whether the transformation is synchronous [SHASH] or asynchronous [AHASH] and the amount of HASH transformations we are registering. You can find the prototypes defined in include/crypto/internal/hash.h:

```
int crypto_register_ahash(struct ahash_alg *alg);

int crypto_register_shash(struct shash_alg *alg);
int crypto_register_shashes(struct shash_alg *algs, int count);
```

The respective counterparts for unregistering the HASH transformation are as follows:

```
int crypto_unregister_ahash(struct ahash_alg *alg);

int crypto_unregister_shash(struct shash_alg *alg);
int crypto_unregister_shashes(struct shash_alg *algs, int count);
```

Cipher Definition With struct shash_alg and ahash_alg

Here are schematics of how these functions are called when operated from other part of the kernel. Note that the .setkey() call might happen before or after any of these schematics happen, but must not happen during any of these are in-flight. Please note that calling .init() followed immediately by .finish() is also a perfectly valid transformation.

```
I)   DATA -----------.
                     v
      .init() -> .update() -> .final()      ! .update() might not be called
                  ^    |         |            at all in this scenario.
                  '----'         '---> HASH

II)  DATA -----------.-----------.
                     v           v
      .init() -> .update() -> .finup()      ! .update() may not be called
                  ^    |         |            at all in this scenario.
                  '----'         '---> HASH

III) DATA -----------.
                     v
                 .digest()                  ! The entire process is handled
                     |                        by the .digest() call.
                     '---------------> HASH
```

Here is a schematic of how the .export()/.import() functions are called when used from another part of the kernel.

```
KEY--.                 DATA--.
     v                       v                  ! .update() may not be called
 .setkey() -> .init() -> .update() -> .export()   at all in this scenario.
                          ^     |         |
                          '-----'         '--> PARTIAL_HASH

----------- other transformations happen here -----------

PARTIAL_HASH--.   DATA1--.
              v          v
          .import -> .update() -> .final()     ! .update() may not be called
                      ^    |         |           at all in this scenario.
                      '----'         '--> HASH1

PARTIAL_HASH--.   DATA2-.
              v         v
          .import -> .finup()
                        |
                        '---------------> HASH2
```

Note that it is perfectly legal to “abandon” a request object: - call .init() and then (as many times) .update() - _not_ call any of .final(), .finup() or .export() at any point in future

In other words implementations should mind the resource allocation and clean-up. No resources related to request objects should remain allocated after a call to .init() or .update(), since there might be no chance to free them.

Specifics Of Asynchronous HASH Transformation

Some of the drivers will want to use the Generic ScatterWalk in case the implementation needs to be fed separate chunks of the scatterlist which contains the input data. The buffer containing the resulting hash will always be properly aligned to .cra_alignmask so there is no need to worry about this.

CHAPTER

FOUR

USER SPACE INTERFACE

Introduction

The concepts of the kernel crypto API visible to kernel space is fully applicable to the user space interface as well. Therefore, the kernel crypto API high level discussion for the in-kernel use cases applies here as well.

The major difference, however, is that user space can only act as a consumer and never as a provider of a transformation or cipher algorithm.

The following covers the user space interface exported by the kernel crypto API. A working example of this description is libkcapi that can be obtained from [1]. That library can be used by user space applications that require cryptographic services from the kernel.

Some details of the in-kernel kernel crypto API aspects do not apply to user space, however. This includes the difference between synchronous and asynchronous invocations. The user space API call is fully synchronous.

[1] http://www.chronox.de/libkcapi.html

User Space API General Remarks

The kernel crypto API is accessible from user space. Currently, the following ciphers are accessible:

- Message digest including keyed message digest (HMAC, CMAC)
- Symmetric ciphers
- AEAD ciphers
- Random Number Generators

The interface is provided via socket type using the type AF_ALG. In addition, the setsockopt option type is SOL_ALG. In case the user space header files do not export these flags yet, use the following macros:

```
#ifndef AF_ALG
#define AF_ALG 38
#endif
#ifndef SOL_ALG
#define SOL_ALG 279
#endif
```

A cipher is accessed with the same name as done for the in-kernel API calls. This includes the generic vs. unique naming schema for ciphers as well as the enforcement of priorities for generic names.

To interact with the kernel crypto API, a socket must be created by the user space application. User space invokes the cipher operation with the send()/write() system call family. The result of the cipher operation is obtained with the read()/recv() system call family.

The following API calls assume that the socket descriptor is already opened by the user space application and discusses only the kernel crypto API specific invocations.

To initialize the socket interface, the following sequence has to be performed by the consumer:

1. Create a socket of type AF_ALG with the struct sockaddr_alg parameter specified below for the different cipher types.
2. Invoke bind with the socket descriptor
3. Invoke accept with the socket descriptor. The accept system call returns a new file descriptor that is to be used to interact with the particular cipher instance. When invoking send/write or recv/read system calls to send data to the kernel or obtain data from the kernel, the file descriptor returned by accept must be used.

In-place Cipher operation

Just like the in-kernel operation of the kernel crypto API, the user space interface allows the cipher operation in-place. That means that the input buffer used for the send/write system call and the output buffer used by the read/recv system call may be one and the same. This is of particular interest for symmetric cipher operations where a copying of the output data to its final destination can be avoided.

If a consumer on the other hand wants to maintain the plaintext and the ciphertext in different memory locations, all a consumer needs to do is to provide different memory pointers for the encryption and decryption operation.

Message Digest API

The message digest type to be used for the cipher operation is selected when invoking the bind syscall. bind requires the caller to provide a filled struct sockaddr data structure. This data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "hash", /* this selects the hash logic in the kernel */
    .salg_name = "sha1" /* this is the cipher name */
};
```

The salg_type value “hash” applies to message digests and keyed message digests. Though, a keyed message digest is referenced by the appropriate salg_name. Please see below for the setsockopt interface that explains how the key can be set for a keyed message digest.

Using the send() system call, the application provides the data that should be processed with the message digest. The send system call allows the following flags to be specified:

- MSG_MORE: If this flag is set, the send system call acts like a message digest update function where the final hash is not yet calculated. If the flag is not set, the send system call calculates the final message digest immediately.

With the recv() system call, the application can read the message digest from the kernel crypto API. If the buffer is too small for the message digest, the flag MSG_TRUNC is set by the kernel.

In order to set a message digest key, the calling application must use the setsockopt() option of ALG_SET_KEY. If the key is not set the HMAC operation is performed without the initial HMAC state change caused by the key.

Symmetric Cipher API

The operation is very similar to the message digest discussion. During initialization, the struct sockaddr data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "skcipher", /* this selects the symmetric cipher */
    .salg_name = "cbc(aes)" /* this is the cipher name */
};
```

Before data can be sent to the kernel using the write/send system call family, the consumer must set the key. The key setting is described with the setsockopt invocation below.

Using the sendmsg() system call, the application provides the data that should be processed for encryption or decryption. In addition, the IV is specified with the data structure provided by the sendmsg() system call.

The sendmsg system call parameter of struct msghdr is embedded into the struct cmsghdr data structure. See recv(2) and cmsg(3) for more information on how the cmsghdr data structure is used together with the send/recv system call family. That cmsghdr data structure holds the following information specified with a separate header instances:

- specification of the cipher operation type with one of these flags:
  - ALG_OP_ENCRYPT - encryption of data
  - ALG_OP_DECRYPT - decryption of data
- specification of the IV information marked with the flag ALG_SET_IV

The send system call family allows the following flag to be specified:

- MSG_MORE: If this flag is set, the send system call acts like a cipher update function where more input data is expected with a subsequent invocation of the send system call.

Note: The kernel reports -EINVAL for any unexpected data. The caller must make sure that all data matches the constraints given in /proc/crypto for the selected cipher.

With the recv() system call, the application can read the result of the cipher operation from the kernel crypto API. The output buffer must be at least as large as to hold all blocks of the encrypted or decrypted data. If the output data size is smaller, only as many blocks are returned that fit into that output buffer size.

AEAD Cipher API

The operation is very similar to the symmetric cipher discussion. During initialization, the struct sockaddr data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "aead", /* this selects the symmetric cipher */
    .salg_name = "gcm(aes)" /* this is the cipher name */
};
```

Before data can be sent to the kernel using the write/send system call family, the consumer must set the key. The key setting is described with the setsockopt invocation below.

In addition, before data can be sent to the kernel using the write/send system call family, the consumer must set the authentication tag size. To set the authentication tag size, the caller must use the setsockopt invocation described below.

Using the sendmsg() system call, the application provides the data that should be processed for encryption or decryption. In addition, the IV is specified with the data structure provided by the sendmsg() system call.

The sendmsg system call parameter of struct msghdr is embedded into the struct cmsghdr data structure. See recv(2) and cmsg(3) for more information on how the cmsghdr data structure is used together with the send/recv system call family. That cmsghdr data structure holds the following information specified with a separate header instances:

- specification of the cipher operation type with one of these flags:
  - ALG_OP_ENCRYPT - encryption of data
  - ALG_OP_DECRYPT - decryption of data
- specification of the IV information marked with the flag ALG_SET_IV
- specification of the associated authentication data (AAD) with the flag ALG_SET_AEAD_ASSOCLEN. The AAD is sent to the kernel together with the plaintext / ciphertext. See below for the memory structure.

The send system call family allows the following flag to be specified:

- MSG_MORE: If this flag is set, the send system call acts like a cipher update function where more input data is expected with a subsequent invocation of the send system call.

Note: The kernel reports -EINVAL for any unexpected data. The caller must make sure that all data matches the constraints given in /proc/crypto for the selected cipher.

With the recv() system call, the application can read the result of the cipher operation from the kernel crypto API. The output buffer must be at least as large as defined with the memory structure below. If the output data size is smaller, the cipher operation is not performed.

The authenticated decryption operation may indicate an integrity error. Such breach in integrity is marked with the -EBADMSG error code.

AEAD Memory Structure

The AEAD cipher operates with the following information that is communicated between user and kernel space as one data stream:

- plaintext or ciphertext
- associated authentication data (AAD)
- authentication tag

The sizes of the AAD and the authentication tag are provided with the sendmsg and setsockopt calls (see there). As the kernel knows the size of the entire data stream, the kernel is now able to calculate the right offsets of the data components in the data stream.

The user space caller must arrange the aforementioned information in the following order:

- AEAD encryption input: AAD || plaintext
- AEAD decryption input: AAD || ciphertext || authentication tag

The output buffer the user space caller provides must be at least as large to hold the following data:

- AEAD encryption output: ciphertext || authentication tag
- AEAD decryption output: plaintext

Random Number Generator API

Again, the operation is very similar to the other APIs. During initialization, the struct sockaddr data structure must be filled as follows:

```
struct sockaddr_alg sa = {
    .salg_family = AF_ALG,
    .salg_type = "rng", /* this selects the symmetric cipher */
    .salg_name = "drbg_nopr_sha256" /* this is the cipher name */
};
```

Depending on the RNG type, the RNG must be seeded. The seed is provided using the setsockopt interface to set the key. For example, the ansi_cprng requires a seed. The DRBGs do not require a seed, but may be seeded.

Using the read()/recvmsg() system calls, random numbers can be obtained. The kernel generates at most 128 bytes in one call. If user space requires more data, multiple calls to read()/recvmsg() must be made.

WARNING: The user space caller may invoke the initially mentioned accept system call multiple times. In this case, the returned file descriptors have the same state.

Zero-Copy Interface

In addition to the send/write/read/recv system call family, the AF_ALG interface can be accessed with the zero-copy interface of splice/vmsplice. As the name indicates, the kernel tries to avoid a copy operation into kernel space.

The zero-copy operation requires data to be aligned at the page boundary. Non-aligned data can be used as well, but may require more operations of the kernel which would defeat the speed gains obtained from the zero-copy interface.

The system-inherent limit for the size of one zero-copy operation is 16 pages. If more data is to be sent to AF_ALG, user space must slice the input into segments with a maximum size of 16 pages.

Zero-copy can be used with the following code example (a complete working example is provided with libkcapi):

```
int pipes[2];

pipe(pipes);
/* input data in iov */
vmsplice(pipes[1], iov, iovlen, SPLICE_F_GIFT);
/* opfd is the file descriptor returned from accept() system call */
splice(pipes[0], NULL, opfd, NULL, ret, 0);
read(opfd, out, outlen);
```

Setsockopt Interface

In addition to the read/recv and send/write system call handling to send and retrieve data subject to the cipher operation, a consumer also needs to set the additional information for the cipher operation. This additional information is set using the setsockopt system call that must be invoked with the file descriptor of the open cipher (i.e. the file descriptor returned by the accept system call).

Each setsockopt invocation must use the level SOL_ALG.

The setsockopt interface allows setting the following data using the mentioned optname:

- ALG_SET_KEY – Setting the key. Key setting is applicable to:
  - the skcipher cipher type (symmetric ciphers)

  - the hash cipher type (keyed message digests)
  - the AEAD cipher type
  - the RNG cipher type to provide the seed

- ALG_SET_AEAD_AUTHSIZE – Setting the authentication tag size for AEAD ciphers. For a encryption operation, the authentication tag of the given size will be generated. For a decryption operation, the provided ciphertext is assumed to contain an authentication tag of the given size (see section about AEAD memory layout below).

User space API example

Please see [1] for libkcapi which provides an easy-to-use wrapper around the aforementioned Netlink kernel interface. [1] also contains a test application that invokes all libkcapi API calls.

[1] http://www.chronox.de/libkcapi.html

CHAPTER

FIVE

CRYPTO ENGINE

Overview

The crypto engine API (CE), is a crypto queue manager.

Requirement

You have to put at start of your tfm_ctx the struct crypto_engine_ctx:

```
struct your_tfm_ctx {
      struct crypto_engine_ctx enginectx;
      ...
};
```

Why: Since CE manage only crypto_async_request, it cannot know the underlying request_type and so have access only on the TFM. So using container_of for accessing __ctx is impossible. Furthermore, the crypto engine cannot know the "struct your_tfm_ctx", so it must assume that crypto_engine_ctx is at start of it.

Order of operations

You have to obtain a struct crypto_engine via crypto_engine_alloc_init(). And start it via crypto_engine_start().

Before transferring any request, you have to fill the enginectx. - prepare_request: (taking a function pointer) If you need to do some processing before doing the request - unprepare_request: (taking a function pointer) Undoing what's done in prepare_request - do_one_request: (taking a function pointer) Do encryption for current request

Note: that those three functions get the crypto_async_request associated with the received request. So your need to get the original request via container_of(areq, struct yourrequesttype_request, base);

When your driver receive a crypto_request, you have to transfer it to the cryptoengine via one of: - crypto_transfer_ablkcipher_request_to_engine() - crypto_transfer_aead_request_to_engine() - crypto_transfer_akcipher_request_to_engine() - crypto_transfer_hash_request_to_engine() - crypto_transfer_skcipher_request_to_engine()

At the end of the request process, a call to one of the following function is needed: - crypto_finalize_ablkcipher_request - crypto_finalize_aead_request - crypto_finalize_akcipher_request - crypto_finalize_hash_request - crypto_finalize_skcipher_request

CHAPTER

SIX

PROGRAMMING INTERFACE

Please note that the kernel crypto API contains the AEAD givcrypt API (crypto_aead_giv* and aead_givcrypt* function calls in include/crypto/aead.h). This API is obsolete and will be removed in the future. To obtain the functionality of an AEAD cipher with internal IV generation, use the IV generator as a regular cipher. For example, rfc4106(gcm(aes)) is the AEAD cipher with external IV generation and seqniv(rfc4106(gcm(aes))) implies that the kernel crypto API generates the IV. Different IV generators are available.

Table of contents

Block Cipher Algorithm Definitions

These data structures define modular crypto algorithm implementations, managed via `crypto_register_alg()` and `crypto_unregister_alg()`.

struct **`ablkcipher_alg`**
asynchronous block cipher definition

Definition

```
struct ablkcipher_alg {
  int (*setkey)(struct crypto_ablkcipher *tfm, const u8 *key, unsigned int keylen);
  int (*encrypt)(struct ablkcipher_request *req);
  int (*decrypt)(struct ablkcipher_request *req);
  int (*givencrypt)(struct skcipher_givcrypt_request *req);
  int (*givdecrypt)(struct skcipher_givcrypt_request *req);
  const char *geniv;
  unsigned int min_keysize;
  unsigned int max_keysize;
  unsigned int ivsize;
};
```

Members

`setkey` Set key for the transformation. This function is used to either program a supplied key into the hardware or store the key in the transformation context for programming it later. Note that this function does modify the transformation context. This function can be called multiple times during the existence of the transformation object, so one must make sure the key is properly reprogrammed into the hardware. This function is also responsible for checking the key length for validity. In case a software fallback was put in place in the **cra_init** call, this function might need to use the fallback if the algorithm doesn't support all of the key sizes.

`encrypt` Encrypt a scatterlist of blocks. This function is used to encrypt the supplied scatterlist containing the blocks of data. The crypto API consumer is responsible for aligning the entries of the scatterlist properly and making sure the chunks are correctly sized. In case a software fallback was put in place in the **cra_init** call, this function might need to use the fallback if the algorithm doesn't support all of the key sizes. In case the key was stored in transformation context, the key might need to be re-programmed into the hardware in this function. This function shall not modify the transformation context, as this function may be called in parallel with the same transformation object.

decrypt Decrypt a single block. This is a reverse counterpart to **encrypt** and the conditions are exactly the same.

givencrypt Update the IV for encryption. With this function, a cipher implementation may provide the function on how to update the IV for encryption.

givdecrypt Update the IV for decryption. This is the reverse of **givencrypt** .

geniv The transformation implementation may use an "IV generator" provided by the kernel crypto API. Several use cases have a predefined approach how IVs are to be updated. For such use cases, the kernel crypto API provides ready-to-use implementations that can be referenced with this variable.

min_keysize Minimum key size supported by the transformation. This is the smallest key length supported by this transformation algorithm. This must be set to one of the pre-defined values as this is not hardware specific. Possible values for this field can be found via git grep "_MIN_KEY_SIZE" include/crypto/

max_keysize Maximum key size supported by the transformation. This is the largest key length supported by this transformation algorithm. This must be set to one of the pre-defined values as this is not hardware specific. Possible values for this field can be found via git grep "_MAX_KEY_SIZE" include/crypto/

ivsize IV size applicable for transformation. The consumer must provide an IV of exactly that size to perform the encrypt or decrypt operation.

Description

All fields except **givencrypt** , **givdecrypt** , **geniv** and **ivsize** are mandatory and must be filled.

struct **blkcipher_alg**
synchronous block cipher definition

Definition

```
struct blkcipher_alg {
  int (*setkey)(struct crypto_tfm *tfm, const u8 *key, unsigned int keylen);
  int (*encrypt)(struct blkcipher_desc *desc,struct scatterlist *dst, struct scatterlist *src, unsigned
  int (*decrypt)(struct blkcipher_desc *desc,struct scatterlist *dst, struct scatterlist *src, unsigned
  const char *geniv;
  unsigned int min_keysize;
  unsigned int max_keysize;
  unsigned int ivsize;
};
```

Members

setkey see struct ablkcipher_alg

encrypt see struct ablkcipher_alg

decrypt see struct ablkcipher_alg

geniv see struct ablkcipher_alg

min_keysize see struct ablkcipher_alg

max_keysize see struct ablkcipher_alg

ivsize see struct ablkcipher_alg

Description

All fields except **geniv** and **ivsize** are mandatory and must be filled.

struct **cipher_alg**
single-block symmetric ciphers definition

Definition

```
struct cipher_alg {
  unsigned int cia_min_keysize;
  unsigned int cia_max_keysize;
  int (*cia_setkey)(struct crypto_tfm *tfm, const u8 *key, unsigned int keylen);
  void (*cia_encrypt)(struct crypto_tfm *tfm, u8 *dst, const u8 *src);
  void (*cia_decrypt)(struct crypto_tfm *tfm, u8 *dst, const u8 *src);
};
```

Members

cia_min_keysize Minimum key size supported by the transformation. This is the smallest key length supported by this transformation algorithm. This must be set to one of the pre-defined values as this is not hardware specific. Possible values for this field can be found via git grep "_MIN_KEY_SIZE" include/crypto/

cia_max_keysize Maximum key size supported by the transformation. This is the largest key length supported by this transformation algorithm. This must be set to one of the pre-defined values as this is not hardware specific. Possible values for this field can be found via git grep "_MAX_KEY_SIZE" include/crypto/

cia_setkey Set key for the transformation. This function is used to either program a supplied key into the hardware or store the key in the transformation context for programming it later. Note that this function does modify the transformation context. This function can be called multiple times during the existence of the transformation object, so one must make sure the key is properly reprogrammed into the hardware. This function is also responsible for checking the key length for validity.

cia_encrypt Encrypt a single block. This function is used to encrypt a single block of data, which must be **cra_blocksize** big. This always operates on a full **cra_blocksize** and it is not possible to encrypt a block of smaller size. The supplied buffers must therefore also be at least of **cra_blocksize** size. Both the input and output buffers are always aligned to **cra_alignmask**. In case either of the input or output buffer supplied by user of the crypto API is not aligned to **cra_alignmask**, the crypto API will re-align the buffers. The re-alignment means that a new buffer will be allocated, the data will be copied into the new buffer, then the processing will happen on the new buffer, then the data will be copied back into the original buffer and finally the new buffer will be freed. In case a software fallback was put in place in the **cra_init** call, this function might need to use the fallback if the algorithm doesn't support all of the key sizes. In case the key was stored in transformation context, the key might need to be re-programmed into the hardware in this function. This function shall not modify the transformation context, as this function may be called in parallel with the same transformation object.

cia_decrypt Decrypt a single block. This is a reverse counterpart to **cia_encrypt**, and the conditions are exactly the same.

Description

All fields are mandatory and must be filled.

struct **crypto_alg**
definition of a cryptograpic cipher algorithm

Definition

```
struct crypto_alg {
  struct list_head cra_list;
  struct list_head cra_users;
  u32 cra_flags;
  unsigned int cra_blocksize;
  unsigned int cra_ctxsize;
  unsigned int cra_alignmask;
  int cra_priority;
  refcount_t cra_refcnt;
  char cra_name[CRYPTO_MAX_ALG_NAME];
  char cra_driver_name[CRYPTO_MAX_ALG_NAME];
  const struct crypto_type *cra_type;
  union {
    struct ablkcipher_alg ablkcipher;
    struct blkcipher_alg blkcipher;
    struct cipher_alg cipher;
    struct compress_alg compress;
  } cra_u;
  int (*cra_init)(struct crypto_tfm *tfm);
  void (*cra_exit)(struct crypto_tfm *tfm);
  void (*cra_destroy)(struct crypto_alg *alg);
  struct module *cra_module;
};
```

Members

`cra_list` internally used

`cra_users` internally used

`cra_flags` Flags describing this transformation. See include/linux/crypto.h CRYPTO_ALG_* flags for the flags which go in here. Those are used for fine-tuning the description of the transformation algorithm.

`cra_blocksize` Minimum block size of this transformation. The size in bytes of the smallest possible unit which can be transformed with this algorithm. The users must respect this value. In case of HASH transformation, it is possible for a smaller block than **cra_blocksize** to be passed to the crypto API for transformation, in case of any other transformation type, an error will be returned upon any attempt to transform smaller than **cra_blocksize** chunks.

`cra_ctxsize` Size of the operational context of the transformation. This value informs the kernel crypto API about the memory size needed to be allocated for the transformation context.

`cra_alignmask` Alignment mask for the input and output data buffer. The data buffer containing the input data for the algorithm must be aligned to this alignment mask. The data buffer for the output data must be aligned to this alignment mask. Note that the Crypto API will do the re-alignment in software, but only under special conditions and there is a performance hit. The re-alignment happens at these occasions for different **cra_u** types: cipher – For both input data and output data buffer; ahash – For output hash destination buf; shash – For output hash destination buf. This is needed on hardware which is flawed by design and cannot pick data from arbitrary addresses.

`cra_priority` Priority of this transformation implementation. In case multiple transformations with same **cra_name** are available to the Crypto API, the kernel will use the one with highest **cra_priority**.

`cra_refcnt` internally used

`cra_name` Generic name (usable by multiple implementations) of the transformation algorithm. This is the name of the transformation itself. This field is used by the kernel when looking up the providers of particular transformation.

`cra_driver_name` Unique name of the transformation provider. This is the name of the provider of the transformation. This can be any arbitrary value, but in the usual case, this contains the name of the chip or provider and the name of the transformation algorithm.

`cra_type` Type of the cryptographic transformation. This is a pointer to struct crypto_type, which implements callbacks common for all transformation types. There are multiple options: `crypto_blkcipher_type`, `crypto_ablkcipher_type`, `crypto_ahash_type`, `crypto_rng_type`. This field might be empty. In that case, there are no common callbacks. This is the case for: cipher, compress, shash.

`cra_u` Callbacks implementing the transformation. This is a union of multiple structures. Depending on the type of transformation selected by **cra_type** and **cra_flags** above, the associated structure must be filled with callbacks. This field might be empty. This is the case for ahash, shash.

`cra_u.ablkcipher` Union member which contains an asynchronous block cipher definition. See **struct ablkcipher_alg**.

`cra_u.blkcipher` Union member which contains a synchronous block cipher definition See **struct blkcipher_alg**.

`cra_u.cipher` Union member which contains a single-block symmetric cipher definition. See **struct cipher_alg**.

`cra_u.compress` Union member which contains a (de)compression algorithm. See **struct compress_alg**.

`cra_init` Initialize the cryptographic transformation object. This function is used to initialize the cryptographic transformation object. This function is called only once at the instantiation time, right after the transformation context was allocated. In case the cryptographic hardware has some special requirements which need to be handled by software, this function shall check for the precise requirement of the transformation and put any software fallbacks in place.

`cra_exit` Deinitialize the cryptographic transformation object. This is a counterpart to **cra_init**, used to remove various changes set in **cra_init**.

`cra_destroy` internally used

`cra_module` Owner of this transformation implementation. Set to THIS_MODULE

Description

The struct crypto_alg describes a generic Crypto API algorithm and is common for all of the transformations. Any variable not documented here shall not be used by a cipher implementation as it is internal to the Crypto API.

Symmetric Key Cipher API

Symmetric key cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_SKCIPHER (listed as type "skcipher" in /proc/crypto).

Asynchronous cipher operations imply that the function invocation for a cipher request returns immediately before the completion of the operation. The cipher request is scheduled as a separate kernel thread and therefore load-balanced on the different CPUs via the process scheduler. To allow the kernel crypto API to inform the caller about the completion of a cipher request, the caller must provide a callback function. That function is invoked with the cipher handle when the request completes.

To support the asynchronous operation, additional information than just the cipher handle must be supplied to the kernel crypto API. That additional information is given by filling in the skcipher_request data structure.

For the symmetric key cipher API, the state is maintained with the tfm cipher handle. A single tfm can be used across multiple calls and in parallel. For asynchronous block cipher calls, context data supplied and only used by the caller can be referenced the request data structure in addition to the IV used for the cipher request. The maintenance of such state information would be important for a crypto driver implementer to have, because when calling the callback function upon completion of the cipher operation, that callback function may need some information about which operation just finished if it invoked multiple in parallel. This state information is unused by the kernel crypto API.

struct crypto_skcipher * **`crypto_alloc_skcipher`**(const char * *alg_name*, u32 *type*, u32 *mask*)
allocate symmetric key cipher handle

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the skcipher cipher

`u32 type` specifies the type of the cipher

`u32 mask` specifies the mask for the cipher

Description

Allocate a cipher handle for an skcipher. The returned struct crypto_skcipher is the cipher handle that is required for any subsequent API invocation for that skcipher.

Return

allocated cipher handle in case of success; `IS_ERR()` is true in case of an error, PTR_ERR() returns the error code.

void **`crypto_free_skcipher`**(struct crypto_skcipher * *tfm*)
zeroize and free cipher handle

Parameters

`struct crypto_skcipher * tfm` cipher handle to be freed

int **`crypto_has_skcipher`**(const char * *alg_name*, u32 *type*, u32 *mask*)
Search for the availability of an skcipher.

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the skcipher

`u32 type` specifies the type of the cipher

`u32 mask` specifies the mask for the cipher

Return

true when the skcipher is known to the kernel crypto API; false otherwise

unsigned int **`crypto_skcipher_ivsize`**(struct crypto_skcipher * *tfm*)
obtain IV size

Parameters

`struct crypto_skcipher * tfm` cipher handle

Description

The size of the IV for the skcipher referenced by the cipher handle is returned. This IV size may be zero if the cipher does not need an IV.

Return

IV size in bytes

unsigned int **`crypto_skcipher_blocksize`**(struct crypto_skcipher * *tfm*)
obtain block size of cipher

Parameters

`struct crypto_skcipher * tfm` cipher handle

Description

The block size for the skcipher referenced with the cipher handle is returned. The caller may use that information to allocate appropriate memory for the data returned by the encryption or decryption operation

Return

block size of cipher

int **`crypto_skcipher_setkey`**(struct crypto_skcipher * *tfm*, const u8 * *key*, unsigned int *keylen*)
set key for cipher

Parameters

`struct crypto_skcipher * tfm` cipher handle

`const u8 * key` buffer holding the key

`unsigned int keylen` length of the key in bytes

Description

The caller provided key is set for the skcipher referenced by the cipher handle.

Note, the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

struct crypto_skcipher * **`crypto_skcipher_reqtfm`**(struct skcipher_request * *req*)
obtain cipher handle from request

Parameters

`struct skcipher_request * req` skcipher_request out of which the cipher handle is to be obtained

Description

Return the crypto_skcipher handle when furnishing an skcipher_request data structure.

Return

crypto_skcipher handle

int **`crypto_skcipher_encrypt`**(struct skcipher_request * *req*)
encrypt plaintext

Parameters

`struct skcipher_request * req` reference to the skcipher_request handle that holds all information needed to perform the cipher operation

Description

Encrypt plaintext data using the skcipher_request handle. That data structure and how it is filled with data is discussed with the skcipher_request_* functions.

Return

0 if the cipher operation was successful; < 0 if an error occurred

int **`crypto_skcipher_decrypt`**(struct skcipher_request * *req*)
decrypt ciphertext

Parameters

`struct skcipher_request * req` reference to the skcipher_request handle that holds all information needed to perform the cipher operation

Description

Decrypt ciphertext data using the skcipher_request handle. That data structure and how it is filled with data is discussed with the skcipher_request_* functions.

Return

0 if the cipher operation was successful; < 0 if an error occurred

Symmetric Key Cipher Request Handle

The skcipher_request data structure contains all pointers to data required for the symmetric key cipher operation. This includes the cipher handle (which can be used by multiple skcipher_request instances), pointer to plaintext and ciphertext, asynchronous callback function, etc. It acts as a handle to the skcipher_request_* API calls in a similar way as skcipher handle to the crypto_skcipher_* API calls.

unsigned int **`crypto_skcipher_reqsize`**(struct crypto_skcipher * *tfm*)
obtain size of the request data structure

Parameters

`struct crypto_skcipher * tfm` cipher handle

Return

number of bytes

void **`skcipher_request_set_tfm`**(struct skcipher_request * *req*, struct crypto_skcipher * *tfm*)
update cipher handle reference in request

Parameters

`struct skcipher_request * req` request handle to be modified

`struct crypto_skcipher * tfm` cipher handle that shall be added to the request handle

Description

Allow the caller to replace the existing skcipher handle in the request data structure with a different one.

struct skcipher_request * **`skcipher_request_alloc`**(struct crypto_skcipher * *tfm*, gfp_t *gfp*)
allocate request data structure

Parameters

`struct crypto_skcipher * tfm` cipher handle to be registered with the request

`gfp_t gfp` memory allocation flag that is handed to kmalloc by the API call.

Description

Allocate the request data structure that must be used with the skcipher encrypt and decrypt API calls. During the allocation, the provided skcipher handle is registered in the request data structure.

Return

allocated request handle in case of success, or NULL if out of memory

void **`skcipher_request_free`**(struct skcipher_request * *req*)
zeroize and free request data structure

Parameters

`struct skcipher_request * req` request data structure cipher handle to be freed

void **`skcipher_request_set_callback`**(struct skcipher_request * *req*, u32 *flags*, crypto_completion_t *compl*, void * *data*)
set asynchronous callback function

Parameters

`struct skcipher_request * req` request handle

`u32 flags` specify zero or an ORing of the flags CRYPTO_TFM_REQ_MAY_BACKLOG the request queue may back log and increase the wait queue beyond the initial maximum size; CRYPTO_TFM_REQ_MAY_SLEEP the request processing may sleep

`crypto_completion_t compl` callback function pointer to be registered with the request handle

`void * data` The data pointer refers to memory that is not used by the kernel crypto API, but provided to the callback function for it to use. Here, the caller can provide a reference to memory the callback function can operate on. As the callback function is invoked asynchronously to the related functionality, it may need to access data structures of the related functionality which can be referenced using this pointer. The callback function can access the memory via the “data” field in the crypto_async_request data structure provided to the callback function.

Description

This function allows setting the callback function that is triggered once the cipher operation completes.

The callback function is registered with the skcipher_request handle and must comply with the following template:

```
void callback_function(struct crypto_async_request *req, int error)
```

void **skcipher_request_set_crypt**(struct skcipher_request * *req*, struct scatterlist * *src*, struct scatterlist * *dst*, unsigned int *cryptlen*, void * *iv*)
set data buffers

Parameters

struct skcipher_request * req request handle

struct scatterlist * src source scatter / gather list

struct scatterlist * dst destination scatter / gather list

unsigned int cryptlen number of bytes to process from **src**

void * iv IV for the cipher operation which must comply with the IV size defined by crypto_skcipher_ivsize

Description

This function allows setting of the source data and destination data scatter / gather lists.

For encryption, the source is treated as the plaintext and the destination is the ciphertext. For a decryption operation, the use is reversed - the source is the ciphertext and the destination is the plaintext.

Single Block Cipher API

The single block cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_CIPHER (listed as type "cipher" in /proc/crypto).

Using the single block cipher API calls, operations with the basic cipher primitive can be implemented. These cipher primitives exclude any block chaining operations including IV handling.

The purpose of this single block cipher API is to support the implementation of templates or other concepts that only need to perform the cipher operation on one block at a time. Templates invoke the underlying cipher primitive block-wise and process either the input or the output data of these cipher operations.

struct crypto_cipher * **crypto_alloc_cipher**(const char * *alg_name*, u32 *type*, u32 *mask*)
allocate single block cipher handle

Parameters

const char * alg_name is the cra_name / name or cra_driver_name / driver name of the single block cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for a single block cipher. The returned struct crypto_cipher is the cipher handle that is required for any subsequent API invocation for that single block cipher.

Return

allocated cipher handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void **`crypto_free_cipher`**(struct crypto_cipher * *tfm*)
zeroize and free the single block cipher handle

Parameters

`struct crypto_cipher * tfm` cipher handle to be freed

int **`crypto_has_cipher`**(const char * *alg_name*, u32 *type*, u32 *mask*)
Search for the availability of a single block cipher

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the single block cipher

`u32 type` specifies the type of the cipher

`u32 mask` specifies the mask for the cipher

Return

true when the single block cipher is known to the kernel crypto API; false otherwise

unsigned int **`crypto_cipher_blocksize`**(struct crypto_cipher * *tfm*)
obtain block size for cipher

Parameters

`struct crypto_cipher * tfm` cipher handle

Description

The block size for the single block cipher referenced with the cipher handle tfm is returned. The caller may use that information to allocate appropriate memory for the data returned by the encryption or decryption operation

Return

block size of cipher

int **`crypto_cipher_setkey`**(struct crypto_cipher * *tfm*, const u8 * *key*, unsigned int *keylen*)
set key for cipher

Parameters

`struct crypto_cipher * tfm` cipher handle

`const u8 * key` buffer holding the key

`unsigned int keylen` length of the key in bytes

Description

The caller provided key is set for the single block cipher referenced by the cipher handle.

Note, the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

void **`crypto_cipher_encrypt_one`**(struct crypto_cipher * *tfm*, u8 * *dst*, const u8 * *src*)
encrypt one block of plaintext

Parameters

`struct crypto_cipher * tfm` cipher handle

`u8 * dst` points to the buffer that will be filled with the ciphertext

`const u8 * src` buffer holding the plaintext to be encrypted

Description

Invoke the encryption operation of one block. The caller must ensure that the plaintext and ciphertext buffers are at least one block in size.

void **`crypto_cipher_decrypt_one`**(struct crypto_cipher * *tfm*, u8 * *dst*, const u8 * *src*)
decrypt one block of ciphertext

Parameters

`struct crypto_cipher * tfm` cipher handle

`u8 * dst` points to the buffer that will be filled with the plaintext

`const u8 * src` buffer holding the ciphertext to be decrypted

Description

Invoke the decryption operation of one block. The caller must ensure that the plaintext and ciphertext buffers are at least one block in size.

Asynchronous Block Cipher API - Deprecated

Asynchronous block cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_ABLKCIPHER (listed as type "ablkcipher" in /proc/crypto).

Asynchronous cipher operations imply that the function invocation for a cipher request returns immediately before the completion of the operation. The cipher request is scheduled as a separate kernel thread and therefore load-balanced on the different CPUs via the process scheduler. To allow the kernel crypto API to inform the caller about the completion of a cipher request, the caller must provide a callback function. That function is invoked with the cipher handle when the request completes.

To support the asynchronous operation, additional information than just the cipher handle must be supplied to the kernel crypto API. That additional information is given by filling in the ablkcipher_request data structure.

For the asynchronous block cipher API, the state is maintained with the tfm cipher handle. A single tfm can be used across multiple calls and in parallel. For asynchronous block cipher calls, context data supplied and only used by the caller can be referenced the request data structure in addition to the IV used for the cipher request. The maintenance of such state information would be important for a crypto driver implementer to have, because when calling the callback function upon completion of the cipher operation, that callback function may need some information about which operation just finished if it invoked multiple in parallel. This state information is unused by the kernel crypto API.

void **`crypto_free_ablkcipher`**(struct crypto_ablkcipher * *tfm*)
zeroize and free cipher handle

Parameters

`struct crypto_ablkcipher * tfm` cipher handle to be freed

int **`crypto_has_ablkcipher`**(const char * *alg_name*, u32 *type*, u32 *mask*)
Search for the availability of an ablkcipher.

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the ablkcipher

`u32 type` specifies the type of the cipher

`u32 mask` specifies the mask for the cipher

Return

true when the ablkcipher is known to the kernel crypto API; false otherwise

unsigned int **`crypto_ablkcipher_ivsize`**(struct crypto_ablkcipher * *tfm*)
obtain IV size

Parameters

`struct crypto_ablkcipher * tfm` cipher handle

Description

The size of the IV for the ablkcipher referenced by the cipher handle is returned. This IV size may be zero if the cipher does not need an IV.

Return

IV size in bytes

unsigned int **`crypto_ablkcipher_blocksize`**(struct crypto_ablkcipher * *tfm*)
obtain block size of cipher

Parameters

`struct crypto_ablkcipher * tfm` cipher handle

Description

The block size for the ablkcipher referenced with the cipher handle is returned. The caller may use that information to allocate appropriate memory for the data returned by the encryption or decryption operation

Return

block size of cipher

int **`crypto_ablkcipher_setkey`**(struct crypto_ablkcipher * *tfm*, const u8 * *key*, unsigned int *keylen*)
set key for cipher

Parameters

`struct crypto_ablkcipher * tfm` cipher handle

`const u8 * key` buffer holding the key

`unsigned int keylen` length of the key in bytes

Description

The caller provided key is set for the ablkcipher referenced by the cipher handle.

Note, the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

struct crypto_ablkcipher * **`crypto_ablkcipher_reqtfm`**(struct ablkcipher_request * *req*)
obtain cipher handle from request

Parameters

`struct ablkcipher_request * req` ablkcipher_request out of which the cipher handle is to be obtained

Description

Return the crypto_ablkcipher handle when furnishing an ablkcipher_request data structure.

Return

crypto_ablkcipher handle

int **`crypto_ablkcipher_encrypt`**(struct ablkcipher_request * *req*)
encrypt plaintext

Parameters

`struct ablkcipher_request * req` reference to the ablkcipher_request handle that holds all information needed to perform the cipher operation

Description

Encrypt plaintext data using the ablkcipher_request handle. That data structure and how it is filled with data is discussed with the ablkcipher_request_* functions.

Return

0 if the cipher operation was successful; < 0 if an error occurred

int **`crypto_ablkcipher_decrypt`**(struct ablkcipher_request * *req*)
decrypt ciphertext

Parameters

`struct ablkcipher_request * req` reference to the ablkcipher_request handle that holds all information needed to perform the cipher operation

Description

Decrypt ciphertext data using the ablkcipher_request handle. That data structure and how it is filled with data is discussed with the ablkcipher_request_* functions.

Return

0 if the cipher operation was successful; < 0 if an error occurred

Asynchronous Cipher Request Handle - Deprecated

The ablkcipher_request data structure contains all pointers to data required for the asynchronous cipher operation. This includes the cipher handle (which can be used by multiple ablkcipher_request instances), pointer to plaintext and ciphertext, asynchronous callback function, etc. It acts as a handle to the ablkcipher_request_* API calls in a similar way as ablkcipher handle to the crypto_ablkcipher_* API calls.

unsigned int **`crypto_ablkcipher_reqsize`**(struct crypto_ablkcipher * *tfm*)
obtain size of the request data structure

Parameters

`struct crypto_ablkcipher * tfm` cipher handle

Return

number of bytes

void **`ablkcipher_request_set_tfm`**(struct ablkcipher_request * *req*, struct crypto_ablkcipher * *tfm*)
update cipher handle reference in request

Parameters

`struct ablkcipher_request * req` request handle to be modified

`struct crypto_ablkcipher * tfm` cipher handle that shall be added to the request handle

Description

Allow the caller to replace the existing ablkcipher handle in the request data structure with a different one.

struct ablkcipher_request * **`ablkcipher_request_alloc`**(struct crypto_ablkcipher * *tfm*, gfp_t *gfp*)
allocate request data structure

Parameters

`struct crypto_ablkcipher * tfm` cipher handle to be registered with the request

gfp_t gfp memory allocation flag that is handed to kmalloc by the API call.

Description

Allocate the request data structure that must be used with the ablkcipher encrypt and decrypt API calls. During the allocation, the provided ablkcipher handle is registered in the request data structure.

Return

allocated request handle in case of success, or NULL if out of memory

void **ablkcipher_request_free**(struct ablkcipher_request * *req*)
zeroize and free request data structure

Parameters

struct ablkcipher_request * req request data structure cipher handle to be freed

void **ablkcipher_request_set_callback**(struct ablkcipher_request * *req*, u32 *flags*, crypto_completion_t *compl*, void * *data*)
set asynchronous callback function

Parameters

struct ablkcipher_request * req request handle

u32 flags specify zero or an ORing of the flags CRYPTO_TFM_REQ_MAY_BACKLOG the request queue may back log and increase the wait queue beyond the initial maximum size; CRYPTO_TFM_REQ_MAY_SLEEP the request processing may sleep

crypto_completion_t compl callback function pointer to be registered with the request handle

void * data The data pointer refers to memory that is not used by the kernel crypto API, but provided to the callback function for it to use. Here, the caller can provide a reference to memory the callback function can operate on. As the callback function is invoked asynchronously to the related functionality, it may need to access data structures of the related functionality which can be referenced using this pointer. The callback function can access the memory via the "data" field in the crypto_async_request data structure provided to the callback function.

Description

This function allows setting the callback function that is triggered once the cipher operation completes.

The callback function is registered with the ablkcipher_request handle and must comply with the following template:

```
void callback_function(struct crypto_async_request *req, int error)
```

void **ablkcipher_request_set_crypt**(struct ablkcipher_request * *req*, struct scatterlist * *src*, struct scatterlist * *dst*, unsigned int *nbytes*, void * *iv*)
set data buffers

Parameters

struct ablkcipher_request * req request handle

struct scatterlist * src source scatter / gather list

struct scatterlist * dst destination scatter / gather list

unsigned int nbytes number of bytes to process from **src**

void * iv IV for the cipher operation which must comply with the IV size defined by crypto_ablkcipher_ivsize

Description

This function allows setting of the source data and destination data scatter / gather lists.

For encryption, the source is treated as the plaintext and the destination is the ciphertext. For a decryption operation, the use is reversed - the source is the ciphertext and the destination is the plaintext.

Synchronous Block Cipher API - Deprecated

The synchronous block cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_BLKCIPHER (listed as type "blkcipher" in /proc/crypto)

Synchronous calls, have a context in the tfm. But since a single tfm can be used in multiple calls and in parallel, this info should not be changeable (unless a lock is used). This applies, for example, to the symmetric key. However, the IV is changeable, so there is an iv field in blkcipher_tfm structure for synchronous blkcipher api. So, its the only state info that can be kept for synchronous calls without using a big lock across a tfm.

The block cipher API allows the use of a complete cipher, i.e. a cipher consisting of a template (a block chaining mode) and a single block cipher primitive (e.g. AES).

The plaintext data buffer and the ciphertext data buffer are pointed to by using scatter/gather lists. The cipher operation is performed on all segments of the provided scatter/gather lists.

The kernel crypto API supports a cipher operation "in-place" which means that the caller may provide the same scatter/gather list for the plaintext and cipher text. After the completion of the cipher operation, the plaintext data is replaced with the ciphertext data in case of an encryption and vice versa for a decryption. The caller must ensure that the scatter/gather lists for the output data point to sufficiently large buffers, i.e. multiples of the block size of the cipher.

struct crypto_blkcipher * **`crypto_alloc_blkcipher`**(const char * *alg_name*, u32 *type*, u32 *mask*)
allocate synchronous block cipher handle

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the blkcipher cipher

`u32 type` specifies the type of the cipher

`u32 mask` specifies the mask for the cipher

Description

Allocate a cipher handle for a block cipher. The returned struct crypto_blkcipher is the cipher handle that is required for any subsequent API invocation for that block cipher.

Return

allocated cipher handle in case of success; `IS_ERR()` is true in case of an error, `PTR_ERR()` returns the error code.

void **`crypto_free_blkcipher`**(struct crypto_blkcipher * *tfm*)
zeroize and free the block cipher handle

Parameters

`struct crypto_blkcipher * tfm` cipher handle to be freed

int **`crypto_has_blkcipher`**(const char * *alg_name*, u32 *type*, u32 *mask*)
Search for the availability of a block cipher

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the block cipher

`u32 type` specifies the type of the cipher

`u32 mask` specifies the mask for the cipher

Return

true when the block cipher is known to the kernel crypto API; false otherwise

const char * **`crypto_blkcipher_name`**(struct crypto_blkcipher * *tfm*)
return the name / cra_name from the cipher handle

Parameters

`struct crypto_blkcipher * tfm` cipher handle

Return

The character string holding the name of the cipher

unsigned int **`crypto_blkcipher_ivsize`**(struct crypto_blkcipher * *tfm*)
obtain IV size

Parameters

`struct crypto_blkcipher * tfm` cipher handle

Description

The size of the IV for the block cipher referenced by the cipher handle is returned. This IV size may be zero if the cipher does not need an IV.

Return

IV size in bytes

unsigned int **`crypto_blkcipher_blocksize`**(struct crypto_blkcipher * *tfm*)
obtain block size of cipher

Parameters

`struct crypto_blkcipher * tfm` cipher handle

Description

The block size for the block cipher referenced with the cipher handle is returned. The caller may use that information to allocate appropriate memory for the data returned by the encryption or decryption operation.

Return

block size of cipher

int **`crypto_blkcipher_setkey`**(struct crypto_blkcipher * *tfm*, const u8 * *key*, unsigned int *keylen*)
set key for cipher

Parameters

`struct crypto_blkcipher * tfm` cipher handle

`const u8 * key` buffer holding the key

`unsigned int keylen` length of the key in bytes

Description

The caller provided key is set for the block cipher referenced by the cipher handle.

Note, the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **`crypto_blkcipher_encrypt`**(struct blkcipher_desc * *desc*, struct scatterlist * *dst*, struct scatterlist * *src*, unsigned int *nbytes*)
encrypt plaintext

Parameters

`struct blkcipher_desc * desc` reference to the block cipher handle with meta data

`struct scatterlist * dst` scatter/gather list that is filled by the cipher operation with the ciphertext

`struct scatterlist * src` scatter/gather list that holds the plaintext

unsigned int nbytes number of bytes of the plaintext to encrypt.

Description

Encrypt plaintext data using the IV set by the caller with a preceding call of crypto_blkcipher_set_iv.

The blkcipher_desc data structure must be filled by the caller and can reside on the stack. The caller must fill desc as follows: desc.tfm is filled with the block cipher handle; desc.flags is filled with either CRYPTO_TFM_REQ_MAY_SLEEP or 0.

Return

0 if the cipher operation was successful; < 0 if an error occurred

int **crypto_blkcipher_encrypt_iv**(struct blkcipher_desc * *desc*, struct scatterlist * *dst*, struct scatterlist * *src*, unsigned int *nbytes*)
encrypt plaintext with dedicated IV

Parameters

struct blkcipher_desc * desc reference to the block cipher handle with meta data

struct scatterlist * dst scatter/gather list that is filled by the cipher operation with the ciphertext

struct scatterlist * src scatter/gather list that holds the plaintext

unsigned int nbytes number of bytes of the plaintext to encrypt.

Description

Encrypt plaintext data with the use of an IV that is solely used for this cipher operation. Any previously set IV is not used.

The blkcipher_desc data structure must be filled by the caller and can reside on the stack. The caller must fill desc as follows: desc.tfm is filled with the block cipher handle; desc.info is filled with the IV to be used for the current operation; desc.flags is filled with either CRYPTO_TFM_REQ_MAY_SLEEP or 0.

Return

0 if the cipher operation was successful; < 0 if an error occurred

int **crypto_blkcipher_decrypt**(struct blkcipher_desc * *desc*, struct scatterlist * *dst*, struct scatterlist * *src*, unsigned int *nbytes*)
decrypt ciphertext

Parameters

struct blkcipher_desc * desc reference to the block cipher handle with meta data

struct scatterlist * dst scatter/gather list that is filled by the cipher operation with the plaintext

struct scatterlist * src scatter/gather list that holds the ciphertext

unsigned int nbytes number of bytes of the ciphertext to decrypt.

Description

Decrypt ciphertext data using the IV set by the caller with a preceding call of crypto_blkcipher_set_iv.

The blkcipher_desc data structure must be filled by the caller as documented for the crypto_blkcipher_encrypt call above.

Return

0 if the cipher operation was successful; < 0 if an error occurred

int **crypto_blkcipher_decrypt_iv**(struct blkcipher_desc * *desc*, struct scatterlist * *dst*, struct scatterlist * *src*, unsigned int *nbytes*)
decrypt ciphertext with dedicated IV

Parameters

struct blkcipher_desc * desc reference to the block cipher handle with meta data

struct scatterlist * dst scatter/gather list that is filled by the cipher operation with the plaintext

struct scatterlist * src scatter/gather list that holds the ciphertext

unsigned int nbytes number of bytes of the ciphertext to decrypt.

Description

Decrypt ciphertext data with the use of an IV that is solely used for this cipher operation. Any previously set IV is not used.

The blkcipher_desc data structure must be filled by the caller as documented for the crypto_blkcipher_encrypt_iv call above.

Return

0 if the cipher operation was successful; < 0 if an error occurred

void **crypto_blkcipher_set_iv**(struct crypto_blkcipher * *tfm*, const u8 * *src*, unsigned int *len*)
set IV for cipher

Parameters

struct crypto_blkcipher * tfm cipher handle

const u8 * src buffer holding the IV

unsigned int len length of the IV in bytes

Description

The caller provided IV is set for the block cipher referenced by the cipher handle.

void **crypto_blkcipher_get_iv**(struct crypto_blkcipher * *tfm*, u8 * *dst*, unsigned int *len*)
obtain IV from cipher

Parameters

struct crypto_blkcipher * tfm cipher handle

u8 * dst buffer filled with the IV

unsigned int len length of the buffer dst

Description

The caller can obtain the IV set for the block cipher referenced by the cipher handle and store it into the user-provided buffer. If the buffer has an insufficient space, the IV is truncated to fit the buffer.

Authenticated Encryption With Associated Data (AEAD) Algorithm Definitions

The AEAD cipher API is used with the ciphers of type CRYPTO_ALG_TYPE_AEAD (listed as type "aead" in /proc/crypto)

The most prominent examples for this type of encryption is GCM and CCM. However, the kernel supports other types of AEAD ciphers which are defined with the following cipher string:

authenc(keyed message digest, block cipher)

For example: authenc(hmac(sha256), cbc(aes))

The example code provided for the symmetric key cipher operation applies here as well. Naturally all *skcipher* symbols must be exchanged the *aead* pendants discussed in the following. In addition, for the AEAD operation, the aead_request_set_ad function must be used to set the pointer to the associated data memory location before performing the encryption or decryption operation. In case of an encryption, the associated data memory is filled during the encryption operation. For decryption, the associated data memory must contain data that is used to verify the integrity of the decrypted data. Another deviation from the asynchronous block cipher operation is that the caller should explicitly check for -EBADMSG of the crypto_aead_decrypt. That error indicates an authentication error, i.e. a breach in the integrity of the message. In essence, that -EBADMSG error code is the key bonus an AEAD cipher has over "standard" block chaining modes.

Memory Structure:

To support the needs of the most prominent user of AEAD ciphers, namely IPSEC, the AEAD ciphers have a special memory layout the caller must adhere to.

The scatter list pointing to the input data must contain:

- for RFC4106 ciphers, the concatenation of associated authentication data || IV || plaintext or ciphertext. Note, the same IV (buffer) is also set with the aead_request_set_crypt call. Note, the API call of aead_request_set_ad must provide the length of the AAD and the IV. The API call of aead_request_set_crypt only points to the size of the input plaintext or ciphertext.
- for "normal" AEAD ciphers, the concatenation of associated authentication data || plaintext or ciphertext.

It is important to note that if multiple scatter gather list entries form the input data mentioned above, the first entry must not point to a NULL buffer. If there is any potential where the AAD buffer can be NULL, the calling code must contain a precaution to ensure that this does not result in the first scatter gather list entry pointing to a NULL buffer.

struct **aead_request**
AEAD request

Definition

```
struct aead_request {
  struct crypto_async_request base;
  unsigned int assoclen;
  unsigned int cryptlen;
  u8 *iv;
  struct scatterlist *src;
  struct scatterlist *dst;
  void *__ctx[];
};
```

Members

base Common attributes for async crypto requests

assoclen Length in bytes of associated data for authentication

cryptlen Length of data to be encrypted or decrypted

iv Initialisation vector

src Source data

dst Destination data

__ctx Start of private context data

struct **aead_alg**
AEAD cipher definition

Definition

```
struct aead_alg {
  int (*setkey)(struct crypto_aead *tfm, const u8 *key, unsigned int keylen);
  int (*setauthsize)(struct crypto_aead *tfm, unsigned int authsize);
  int (*encrypt)(struct aead_request *req);
  int (*decrypt)(struct aead_request *req);
  int (*init)(struct crypto_aead *tfm);
  void (*exit)(struct crypto_aead *tfm);
  const char *geniv;
  unsigned int ivsize;
  unsigned int maxauthsize;
  unsigned int chunksize;
  struct crypto_alg base;
};
```

Members

`setkey` see struct skcipher_alg

`setauthsize` Set authentication size for the AEAD transformation. This function is used to specify the consumer requested size of the authentication tag to be either generated by the transformation during encryption or the size of the authentication tag to be supplied during the decryption operation. This function is also responsible for checking the authentication tag size for validity.

`encrypt` see struct skcipher_alg

`decrypt` see struct skcipher_alg

`init` Initialize the cryptographic transformation object. This function is used to initialize the cryptographic transformation object. This function is called only once at the instantiation time, right after the transformation context was allocated. In case the cryptographic hardware has some special requirements which need to be handled by software, this function shall check for the precise requirement of the transformation and put any software fallbacks in place.

`exit` Deinitialize the cryptographic transformation object. This is a counterpart to **init**, used to remove various changes set in **init**.

`geniv` see struct skcipher_alg

`ivsize` see struct skcipher_alg

`maxauthsize` Set the maximum authentication tag size supported by the transformation. A transformation may support smaller tag sizes. As the authentication tag is a message digest to ensure the integrity of the encrypted data, a consumer typically wants the largest authentication tag possible as defined by this variable.

`chunksize` see struct skcipher_alg

`base` Definition of a generic crypto cipher algorithm.

Description

All fields except **ivsize** is mandatory and must be filled.

Authenticated Encryption With Associated Data (AEAD) Cipher API

struct crypto_aead * **`crypto_alloc_aead`**(const char * *alg_name*, u32 *type*, u32 *mask*)
allocate AEAD cipher handle

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the AEAD cipher

`u32 type` specifies the type of the cipher

`u32 mask` specifies the mask for the cipher

Description

Allocate a cipher handle for an AEAD. The returned struct crypto_aead is the cipher handle that is required for any subsequent API invocation for that AEAD.

Return

allocated cipher handle in case of success; `IS_ERR()` is true in case of an error, PTR_ERR() returns the error code.

void **`crypto_free_aead`**(struct crypto_aead * *tfm*)
zeroize and free aead handle

Parameters

`struct crypto_aead * tfm` cipher handle to be freed

unsigned int **`crypto_aead_ivsize`**(struct crypto_aead * *tfm*)
obtain IV size

Parameters

`struct crypto_aead * tfm` cipher handle

Description

The size of the IV for the aead referenced by the cipher handle is returned. This IV size may be zero if the cipher does not need an IV.

Return

IV size in bytes

unsigned int **`crypto_aead_authsize`**(struct crypto_aead * *tfm*)
obtain maximum authentication data size

Parameters

`struct crypto_aead * tfm` cipher handle

Description

The maximum size of the authentication data for the AEAD cipher referenced by the AEAD cipher handle is returned. The authentication data size may be zero if the cipher implements a hard-coded maximum.

The authentication data may also be known as "tag value".

Return

authentication data size / tag size in bytes

unsigned int **`crypto_aead_blocksize`**(struct crypto_aead * *tfm*)
obtain block size of cipher

Parameters

`struct crypto_aead * tfm` cipher handle

Description

The block size for the AEAD referenced with the cipher handle is returned. The caller may use that information to allocate appropriate memory for the data returned by the encryption or decryption operation

Return

block size of cipher

int **`crypto_aead_setkey`**(struct crypto_aead * *tfm*, const u8 * *key*, unsigned int *keylen*)
set key for cipher

Parameters

`struct crypto_aead * tfm` cipher handle

`const u8 * key` buffer holding the key

`unsigned int keylen` length of the key in bytes

Description

The caller provided key is set for the AEAD referenced by the cipher handle.

Note, the key length determines the cipher type. Many block ciphers implement different cipher modes depending on the key size, such as AES-128 vs AES-192 vs. AES-256. When providing a 16 byte key for an AES cipher handle, AES-128 is performed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **crypto_aead_setauthsize**(struct crypto_aead * *tfm*, unsigned int *authsize*)
set authentication data size

Parameters

struct crypto_aead * tfm cipher handle

unsigned int authsize size of the authentication data / tag in bytes

Description

Set the authentication data size / tag size. AEAD requires an authentication tag (or MAC) in addition to the associated data.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **crypto_aead_encrypt**(struct *aead_request* * *req*)
encrypt plaintext

Parameters

struct aead_request * req reference to the aead_request handle that holds all information needed to perform the cipher operation

Description

Encrypt plaintext data using the aead_request handle. That data structure and how it is filled with data is discussed with the aead_request_* functions.

IMPORTANT NOTE The encryption operation creates the authentication data / tag. That data is concatenated with the created ciphertext. The ciphertext memory size is therefore the given number of block cipher blocks + the size defined by the crypto_aead_setauthsize invocation. The caller must ensure that sufficient memory is available for the ciphertext and the authentication tag.

Return

0 if the cipher operation was successful; < 0 if an error occurred

int **crypto_aead_decrypt**(struct *aead_request* * *req*)
decrypt ciphertext

Parameters

struct aead_request * req reference to the ablkcipher_request handle that holds all information needed to perform the cipher operation

Description

Decrypt ciphertext data using the aead_request handle. That data structure and how it is filled with data is discussed with the aead_request_* functions.

IMPORTANT NOTE The caller must concatenate the ciphertext followed by the authentication data / tag. That authentication data / tag must have the size defined by the crypto_aead_setauthsize invocation.

Return

0 if the cipher operation was successful; -EBADMSG: The AEAD cipher operation performs the authentication of the data during the decryption operation. Therefore, the function returns this error if the authentication of the ciphertext was unsuccessful (i.e. the integrity of the ciphertext or the associated data was violated); < 0 if an error occurred.

Asynchronous AEAD Request Handle

The aead_request data structure contains all pointers to data required for the AEAD cipher operation. This includes the cipher handle (which can be used by multiple aead_request instances), pointer to plaintext and ciphertext, asynchronous callback function, etc. It acts as a handle to the aead_request_* API calls in a similar way as AEAD handle to the crypto_aead_* API calls.

unsigned int **crypto_aead_reqsize**(struct crypto_aead * *tfm*)
obtain size of the request data structure

Parameters

struct crypto_aead * tfm cipher handle

Return

number of bytes

void **aead_request_set_tfm**(struct *aead_request* * *req*, struct crypto_aead * *tfm*)
update cipher handle reference in request

Parameters

struct aead_request * req request handle to be modified

struct crypto_aead * tfm cipher handle that shall be added to the request handle

Description

Allow the caller to replace the existing aead handle in the request data structure with a different one.

struct *aead_request* * **aead_request_alloc**(struct crypto_aead * *tfm*, gfp_t *gfp*)
allocate request data structure

Parameters

struct crypto_aead * tfm cipher handle to be registered with the request

gfp_t gfp memory allocation flag that is handed to kmalloc by the API call.

Description

Allocate the request data structure that must be used with the AEAD encrypt and decrypt API calls. During the allocation, the provided aead handle is registered in the request data structure.

Return

allocated request handle in case of success, or NULL if out of memory

void **aead_request_free**(struct *aead_request* * *req*)
zeroize and free request data structure

Parameters

struct aead_request * req request data structure cipher handle to be freed

void **aead_request_set_callback**(struct *aead_request* * *req*, u32 *flags*, crypto_completion_t *compl*, void * *data*)
set asynchronous callback function

Parameters

struct aead_request * req request handle

u32 flags specify zero or an ORing of the flags CRYPTO_TFM_REQ_MAY_BACKLOG the request queue may back log and increase the wait queue beyond the initial maximum size; CRYPTO_TFM_REQ_MAY_SLEEP the request processing may sleep

crypto_completion_t compl callback function pointer to be registered with the request handle

void * data The data pointer refers to memory that is not used by the kernel crypto API, but provided to the callback function for it to use. Here, the caller can provide a reference to memory the callback function can operate on. As the callback function is invoked asynchronously to the related functionality, it may need to access data structures of the related functionality which can be referenced using this pointer. The callback function can access the memory via the "data" field in the crypto_async_request data structure provided to the callback function.

Description

Setting the callback function that is triggered once the cipher operation completes

The callback function is registered with the aead_request handle and must comply with the following template:

```
void callback_function(struct crypto_async_request *req, int error)
```

void **aead_request_set_crypt**(struct *aead_request* * *req*, struct scatterlist * *src*, struct scatterlist * *dst*, unsigned int *cryptlen*, u8 * *iv*)
set data buffers

Parameters

struct aead_request * req request handle

struct scatterlist * src source scatter / gather list

struct scatterlist * dst destination scatter / gather list

unsigned int cryptlen number of bytes to process from **src**

u8 * iv IV for the cipher operation which must comply with the IV size defined by *crypto_aead_ivsize()*

Description

Setting the source data and destination data scatter / gather lists which hold the associated data concatenated with the plaintext or ciphertext. See below for the authentication tag.

For encryption, the source is treated as the plaintext and the destination is the ciphertext. For a decryption operation, the use is reversed - the source is the ciphertext and the destination is the plaintext.

The memory structure for cipher operation has the following structure:

- AEAD encryption input: assoc data || plaintext
- AEAD encryption output: assoc data || cipherntext || auth tag
- AEAD decryption input: assoc data || ciphertext || auth tag
- AEAD decryption output: assoc data || plaintext

Albeit the kernel requires the presence of the AAD buffer, however, the kernel does not fill the AAD buffer in the output case. If the caller wants to have that data buffer filled, the caller must either use an in-place cipher operation (i.e. same memory location for input/output memory location).

void **aead_request_set_ad**(struct *aead_request* * *req*, unsigned int *assoclen*)
set associated data information

Parameters

struct aead_request * req request handle

unsigned int assoclen number of bytes in associated data

Description

Setting the AD information. This function sets the length of the associated data.

Message Digest Algorithm Definitions

These data structures define modular message digest algorithm implementations, managed via `crypto_register_ahash()`, `crypto_register_shash()`, `crypto_unregister_ahash()` and `crypto_unregister_shash()`.

struct **`hash_alg_common`**
define properties of message digest

Definition

```
struct hash_alg_common {
  unsigned int digestsize;
  unsigned int statesize;
  struct crypto_alg base;
};
```

Members

`digestsize` Size of the result of the transformation. A buffer of this size must be available to the **final** and **finup** calls, so they can store the resulting hash into it. For various predefined sizes, search include/crypto/ using git grep _DIGEST_SIZE include/crypto.

`statesize` Size of the block for partial state of the transformation. A buffer of this size must be passed to the **export** function as it will save the partial state of the transformation into it. On the other side, the **import** function will load the state from a buffer of this size as well.

`base` Start of data structure of cipher algorithm. The common data structure of crypto_alg contains information common to all ciphers. The hash_alg_common data structure now adds the hash-specific information.

struct **`ahash_alg`**
asynchronous message digest definition

Definition

```
struct ahash_alg {
  int (*init)(struct ahash_request *req);
  int (*update)(struct ahash_request *req);
  int (*final)(struct ahash_request *req);
  int (*finup)(struct ahash_request *req);
  int (*digest)(struct ahash_request *req);
  int (*export)(struct ahash_request *req, void *out);
  int (*import)(struct ahash_request *req, const void *in);
  int (*setkey)(struct crypto_ahash *tfm, const u8 *key, unsigned int keylen);
  struct hash_alg_common halg;
};
```

Members

`init` [mandatory] Initialize the transformation context. Intended only to initialize the state of the HASH transformation at the beginning. This shall fill in the internal structures used during the entire duration of the whole transformation. No data processing happens at this point. Driver code implementation must not use req->result.

`update` [mandatory] Push a chunk of data into the driver for transformation. This function actually pushes blocks of data from upper layers into the driver, which then passes those to the hardware as seen fit. This function must not finalize the HASH transformation by calculating the final message digest as this only adds more data into the transformation. This function shall not modify the transformation context, as this function may be called in parallel with the same transformation object. Data processing can happen synchronously [SHASH] or asynchronously [AHASH] at this point. Driver must not use req->result.

`final` [mandatory] Retrieve result from the driver. This function finalizes the transformation and retrieves the resulting hash from the driver and pushes it back to upper layers. No data processing happens at this point unless hardware requires it to finish the transformation (then the data buffered by the device driver is processed).

finup [optional] Combination of **update** and **final**. This function is effectively a combination of **update** and **final** calls issued in sequence. As some hardware cannot do **update** and **final** separately, this callback was added to allow such hardware to be used at least by IPsec. Data processing can happen synchronously [SHASH] or asynchronously [AHASH] at this point.

digest Combination of **init** and **update** and **final**. This function effectively behaves as the entire chain of operations, **init**, **update** and **final** issued in sequence. Just like **finup**, this was added for hardware which cannot do even the **finup**, but can only do the whole transformation in one run. Data processing can happen synchronously [SHASH] or asynchronously [AHASH] at this point.

export Export partial state of the transformation. This function dumps the entire state of the ongoing transformation into a provided block of data so it can be **import** 'ed back later on. This is useful in case you want to save partial result of the transformation after processing certain amount of data and reload this partial result multiple times later on for multiple re-use. No data processing happens at this point. Driver must not use req->result.

import Import partial state of the transformation. This function loads the entire state of the ongoing transformation from a provided block of data so the transformation can continue from this point onward. No data processing happens at this point. Driver must not use req->result.

setkey Set optional key used by the hashing algorithm. Intended to push optional key used by the hashing algorithm from upper layers into the driver. This function can store the key in the transformation context or can outright program it into the hardware. In the former case, one must be careful to program the key into the hardware at appropriate time and one must be careful that .:c:func:*setkey()* can be called multiple times during the existence of the transformation object. Not all hashing algorithms do implement this function as it is only needed for keyed message digests. SHAx/MDx/CRCx do NOT implement this function. HMAC(MDx)/HMAC(SHAx)/CMAC(AES) do implement this function. This function must be called before any other of the **init**, **update**, **final**, **finup**, **digest** is called. No data processing happens at this point.

halg see struct hash_alg_common

struct **shash_alg**
synchronous message digest definition

Definition

```
struct shash_alg {
  int (*init)(struct shash_desc *desc);
  int (*update)(struct shash_desc *desc, const u8 *data, unsigned int len);
  int (*final)(struct shash_desc *desc, u8 *out);
  int (*finup)(struct shash_desc *desc, const u8 *data, unsigned int len, u8 *out);
  int (*digest)(struct shash_desc *desc, const u8 *data, unsigned int len, u8 *out);
  int (*export)(struct shash_desc *desc, void *out);
  int (*import)(struct shash_desc *desc, const void *in);
  int (*setkey)(struct crypto_shash *tfm, const u8 *key, unsigned int keylen);
  unsigned int descsize;
  unsigned int digestsize ;
  unsigned int statesize;
  struct crypto_alg base;
};
```

Members

init see struct ahash_alg

update see struct ahash_alg

final see struct ahash_alg

finup see struct ahash_alg

digest see struct ahash_alg

export see struct ahash_alg

import see struct ahash_alg

setkey see struct ahash_alg

descsize Size of the operational state for the message digest. This state size is the memory size that needs to be allocated for shash_desc.__ctx

digestsize see struct ahash_alg

statesize see struct ahash_alg

base internally used

Asynchronous Message Digest API

The asynchronous message digest API is used with the ciphers of type CRYPTO_ALG_TYPE_AHASH (listed as type "ahash" in /proc/crypto)

The asynchronous cipher operation discussion provided for the CRYPTO_ALG_TYPE_ABLKCIPHER API applies here as well.

struct crypto_ahash * **crypto_alloc_ahash**(const char * *alg_name*, u32 *type*, u32 *mask*)
allocate ahash cipher handle

Parameters

const char * alg_name is the cra_name / name or cra_driver_name / driver name of the ahash cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for an ahash. The returned struct crypto_ahash is the cipher handle that is required for any subsequent API invocation for that ahash.

Return

allocated cipher handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void **crypto_free_ahash**(struct crypto_ahash * *tfm*)
zeroize and free the ahash handle

Parameters

struct crypto_ahash * tfm cipher handle to be freed

unsigned int **crypto_ahash_digestsize**(struct crypto_ahash * *tfm*)
obtain message digest size

Parameters

struct crypto_ahash * tfm cipher handle

Description

The size for the message digest created by the message digest cipher referenced with the cipher handle is returned.

Return

message digest size of cipher

unsigned int **crypto_ahash_statesize**(struct crypto_ahash * *tfm*)
obtain size of the ahash state

Parameters

`struct crypto_ahash * tfm` cipher handle

Description

Return the size of the ahash state. With the *`crypto_ahash_export()`* function, the caller can export the state into a buffer whose size is defined with this function.

Return

size of the ahash state

struct crypto_ahash * **`crypto_ahash_reqtfm`**(struct ahash_request * *req*)
obtain cipher handle from request

Parameters

`struct ahash_request * req` asynchronous request handle that contains the reference to the ahash cipher handle

Description

Return the ahash cipher handle that is registered with the asynchronous request handle ahash_request.

Return

ahash cipher handle

unsigned int **`crypto_ahash_reqsize`**(struct crypto_ahash * *tfm*)
obtain size of the request data structure

Parameters

`struct crypto_ahash * tfm` cipher handle

Return

size of the request data

int **`crypto_ahash_setkey`**(struct crypto_ahash * *tfm*, const u8 * *key*, unsigned int *keylen*)
set key for cipher handle

Parameters

`struct crypto_ahash * tfm` cipher handle

`const u8 * key` buffer holding the key

`unsigned int keylen` length of the key in bytes

Description

The caller provided key is set for the ahash cipher. The cipher handle must point to a keyed hash in order for this function to succeed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **`crypto_ahash_finup`**(struct ahash_request * *req*)
update and finalize message digest

Parameters

`struct ahash_request * req` reference to the ahash_request handle that holds all information needed to perform the cipher operation

Description

This function is a “short-hand” for the function calls of crypto_ahash_update and crypto_ahash_final. The parameters have the same meaning as discussed for those separate functions.

Return

see *crypto_ahash_final()*

int **crypto_ahash_final**(struct ahash_request * *req*)
calculate message digest

Parameters

struct ahash_request * req reference to the ahash_request handle that holds all information needed to perform the cipher operation

Description

Finalize the message digest operation and create the message digest based on all data added to the cipher handle. The message digest is placed into the output buffer registered with the ahash_request handle.

Return

0 if the message digest was successfully calculated; -EINPROGRESS if data is feeded into hardware (DMA) or queued for later; -EBUSY if queue is full and request should be resubmitted later; other < 0 if an error occurred

int **crypto_ahash_digest**(struct ahash_request * *req*)
calculate message digest for a buffer

Parameters

struct ahash_request * req reference to the ahash_request handle that holds all information needed to perform the cipher operation

Description

This function is a "short-hand" for the function calls of crypto_ahash_init, crypto_ahash_update and crypto_ahash_final. The parameters have the same meaning as discussed for those separate three functions.

Return

see *crypto_ahash_final()*

int **crypto_ahash_export**(struct ahash_request * *req*, void * *out*)
extract current message digest state

Parameters

struct ahash_request * req reference to the ahash_request handle whose state is exported

void * out output buffer of sufficient size that can hold the hash state

Description

This function exports the hash state of the ahash_request handle into the caller-allocated output buffer out which must have sufficient size (e.g. by calling *crypto_ahash_statesize()*).

Return

0 if the export was successful; < 0 if an error occurred

int **crypto_ahash_import**(struct ahash_request * *req*, const void * *in*)
import message digest state

Parameters

struct ahash_request * req reference to ahash_request handle the state is imported into

const void * in buffer holding the state

Description

This function imports the hash state into the ahash_request handle from the input buffer. That buffer should have been generated with the crypto_ahash_export function.

Return

0 if the import was successful; < 0 if an error occurred

int **crypto_ahash_init**(struct ahash_request * *req*)
(re)initialize message digest handle

Parameters

struct ahash_request * req ahash_request handle that already is initialized with all necessary data using the ahash_request_* API functions

Description

The call (re-)initializes the message digest referenced by the ahash_request handle. Any potentially existing state created by previous operations is discarded.

Return

see *crypto_ahash_final()*

Asynchronous Hash Request Handle

The ahash_request data structure contains all pointers to data required for the asynchronous cipher operation. This includes the cipher handle (which can be used by multiple ahash_request instances), pointer to plaintext and the message digest output buffer, asynchronous callback function, etc. It acts as a handle to the ahash_request_* API calls in a similar way as ahash handle to the crypto_ahash_* API calls.

void **ahash_request_set_tfm**(struct ahash_request * *req*, struct crypto_ahash * *tfm*)
update cipher handle reference in request

Parameters

struct ahash_request * req request handle to be modified

struct crypto_ahash * tfm cipher handle that shall be added to the request handle

Description

Allow the caller to replace the existing ahash handle in the request data structure with a different one.

struct ahash_request * **ahash_request_alloc**(struct crypto_ahash * *tfm*, gfp_t *gfp*)
allocate request data structure

Parameters

struct crypto_ahash * tfm cipher handle to be registered with the request

gfp_t gfp memory allocation flag that is handed to kmalloc by the API call.

Description

Allocate the request data structure that must be used with the ahash message digest API calls. During the allocation, the provided ahash handle is registered in the request data structure.

Return

allocated request handle in case of success, or NULL if out of memory

void **ahash_request_free**(struct ahash_request * *req*)
zeroize and free the request data structure

Parameters

struct ahash_request * req request data structure cipher handle to be freed

void **ahash_request_set_callback**(struct ahash_request * *req*, u32 *flags*, crypto_completion_t *compl*, void * *data*)
set asynchronous callback function

Parameters

`struct ahash_request * req` request handle

`u32 flags` specify zero or an ORing of the flags CRYPTO_TFM_REQ_MAY_BACKLOG the request queue may back log and increase the wait queue beyond the initial maximum size; CRYPTO_TFM_REQ_MAY_SLEEP the request processing may sleep

`crypto_completion_t compl` callback function pointer to be registered with the request handle

`void * data` The data pointer refers to memory that is not used by the kernel crypto API, but provided to the callback function for it to use. Here, the caller can provide a reference to memory the callback function can operate on. As the callback function is invoked asynchronously to the related functionality, it may need to access data structures of the related functionality which can be referenced using this pointer. The callback function can access the memory via the "data" field in the `crypto_async_request` data structure provided to the callback function.

Description

This function allows setting the callback function that is triggered once the cipher operation completes.

The callback function is registered with the `ahash_request` handle and must comply with the following template:

```
void callback_function(struct crypto_async_request *req, int error)
```

void **`ahash_request_set_crypt`**(struct ahash_request * *req*, struct scatterlist * *src*, u8 * *result*, unsigned int *nbytes*)
set data buffers

Parameters

`struct ahash_request * req` ahash_request handle to be updated

`struct scatterlist * src` source scatter/gather list

`u8 * result` buffer that is filled with the message digest – the caller must ensure that the buffer has sufficient space by, for example, calling *`crypto_ahash_digestsize()`*

`unsigned int nbytes` number of bytes to process from the source scatter/gather list

Description

By using this call, the caller references the source scatter/gather list. The source scatter/gather list points to the data the message digest is to be calculated for.

Synchronous Message Digest API

The synchronous message digest API is used with the ciphers of type CRYPTO_ALG_TYPE_SHASH (listed as type "shash" in /proc/crypto)

The message digest API is able to maintain state information for the caller.

The synchronous message digest API can store user-related context in in its shash_desc request data structure.

struct crypto_shash * **`crypto_alloc_shash`**(const char * *alg_name*, u32 *type*, u32 *mask*)
allocate message digest handle

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the message digest cipher

`u32 type` specifies the type of the cipher

`u32 mask` specifies the mask for the cipher

Description

Allocate a cipher handle for a message digest. The returned `struct crypto_shash` is the cipher handle that is required for any subsequent API invocation for that message digest.

Return

allocated cipher handle in case of success; `IS_ERR()` is true in case of an error, PTR_ERR() returns the error code.

void **`crypto_free_shash`**(struct crypto_shash * *tfm*)
zeroize and free the message digest handle

Parameters

`struct crypto_shash * tfm` cipher handle to be freed

unsigned int **`crypto_shash_blocksize`**(struct crypto_shash * *tfm*)
obtain block size for cipher

Parameters

`struct crypto_shash * tfm` cipher handle

Description

The block size for the message digest cipher referenced with the cipher handle is returned.

Return

block size of cipher

unsigned int **`crypto_shash_digestsize`**(struct crypto_shash * *tfm*)
obtain message digest size

Parameters

`struct crypto_shash * tfm` cipher handle

Description

The size for the message digest created by the message digest cipher referenced with the cipher handle is returned.

Return

digest size of cipher

unsigned int **`crypto_shash_descsize`**(struct crypto_shash * *tfm*)
obtain the operational state size

Parameters

`struct crypto_shash * tfm` cipher handle

Description

The size of the operational state the cipher needs during operation is returned for the hash referenced with the cipher handle. This size is required to calculate the memory requirements to allow the caller allocating sufficient memory for operational state.

The operational state is defined with struct shash_desc where the size of that data structure is to be calculated as sizeof(struct shash_desc) + crypto_shash_descsize(alg)

Return

size of the operational state

int **`crypto_shash_setkey`**(struct crypto_shash * *tfm*, const u8 * *key*, unsigned int *keylen*)
set key for message digest

Parameters

`struct crypto_shash * tfm` cipher handle

`const u8 * key` buffer holding the key

`unsigned int keylen` length of the key in bytes

Description

The caller provided key is set for the keyed message digest cipher. The cipher handle must point to a keyed message digest cipher in order for this function to succeed.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **`crypto_shash_digest`**(struct shash_desc * *desc*, const u8 * *data*, unsigned int *len*, u8 * *out*)
calculate message digest for buffer

Parameters

`struct shash_desc * desc` see *`crypto_shash_final()`*

`const u8 * data` see *`crypto_shash_update()`*

`unsigned int len` see *`crypto_shash_update()`*

`u8 * out` see *`crypto_shash_final()`*

Description

This function is a "short-hand" for the function calls of crypto_shash_init, crypto_shash_update and crypto_shash_final. The parameters have the same meaning as discussed for those separate three functions.

Return

0 if the message digest creation was successful; < 0 if an error occurred

int **`crypto_shash_export`**(struct shash_desc * *desc*, void * *out*)
extract operational state for message digest

Parameters

`struct shash_desc * desc` reference to the operational state handle whose state is exported

`void * out` output buffer of sufficient size that can hold the hash state

Description

This function exports the hash state of the operational state handle into the caller-allocated output buffer out which must have sufficient size (e.g. by calling crypto_shash_descsize).

Return

0 if the export creation was successful; < 0 if an error occurred

int **`crypto_shash_import`**(struct shash_desc * *desc*, const void * *in*)
import operational state

Parameters

`struct shash_desc * desc` reference to the operational state handle the state imported into

`const void * in` buffer holding the state

Description

This function imports the hash state into the operational state handle from the input buffer. That buffer should have been generated with the crypto_ahash_export function.

Return

0 if the import was successful; < 0 if an error occurred

int **`crypto_shash_init`**(struct shash_desc * *desc*)
(re)initialize message digest

Parameters

`struct shash_desc * desc` operational state handle that is already filled

Description

The call (re-)initializes the message digest referenced by the operational state handle. Any potentially existing state created by previous operations is discarded.

Return

0 if the message digest initialization was successful; < 0 if an error occurred

int **`crypto_shash_update`**(struct shash_desc * *desc*, const u8 * *data*, unsigned int *len*)
add data to message digest for processing

Parameters

`struct shash_desc * desc` operational state handle that is already initialized

`const u8 * data` input data to be added to the message digest

`unsigned int len` length of the input data

Description

Updates the message digest state of the operational state handle.

Return

0 if the message digest update was successful; < 0 if an error occurred

int **`crypto_shash_final`**(struct shash_desc * *desc*, u8 * *out*)
calculate message digest

Parameters

`struct shash_desc * desc` operational state handle that is already filled with data

`u8 * out` output buffer filled with the message digest

Description

Finalize the message digest operation and create the message digest based on all data added to the cipher handle. The message digest is placed into the output buffer. The caller must ensure that the output buffer is large enough by using crypto_shash_digestsize.

Return

0 if the message digest creation was successful; < 0 if an error occurred

int **`crypto_shash_finup`**(struct shash_desc * *desc*, const u8 * *data*, unsigned int *len*, u8 * *out*)
calculate message digest of buffer

Parameters

`struct shash_desc * desc` see *`crypto_shash_final()`*

`const u8 * data` see *`crypto_shash_update()`*

`unsigned int len` see *`crypto_shash_update()`*

`u8 * out` see *`crypto_shash_final()`*

Description

This function is a "short-hand" for the function calls of crypto_shash_update and crypto_shash_final. The parameters have the same meaning as discussed for those separate functions.

Return

0 if the message digest creation was successful; < 0 if an error occurred

Random Number Algorithm Definitions

struct **rng_alg**
random number generator definition

Definition

```
struct rng_alg {
  int (*generate)(struct crypto_rng *tfm,const u8 *src, unsigned int slen, u8 *dst, unsigned int dlen);
  int (*seed)(struct crypto_rng *tfm, const u8 *seed, unsigned int slen);
  void (*set_ent)(struct crypto_rng *tfm, const u8 *data, unsigned int len);
  unsigned int seedsize;
  struct crypto_alg base;
};
```

Members

generate The function defined by this variable obtains a random number. The random number generator transform must generate the random number out of the context provided with this call, plus any additional data if provided to the call.

seed Seed or reseed the random number generator. With the invocation of this function call, the random number generator shall become ready for generation. If the random number generator requires a seed for setting up a new state, the seed must be provided by the consumer while invoking this function. The required size of the seed is defined with **seedsize** .

set_ent Set entropy that would otherwise be obtained from entropy source. Internal use only.

seedsize The seed size required for a random number generator initialization defined with this variable. Some random number generators does not require a seed as the seeding is implemented internally without the need of support by the consumer. In this case, the seed size is set to zero.

base Common crypto API algorithm data structure.

Crypto API Random Number API

The random number generator API is used with the ciphers of type CRYPTO_ALG_TYPE_RNG (listed as type "rng" in /proc/crypto)

struct crypto_rng * **crypto_alloc_rng**(const char * *alg_name*, u32 *type*, u32 *mask*)

- allocate RNG handle

Parameters

const char * alg_name is the cra_name / name or cra_driver_name / driver name of the message digest cipher

u32 type specifies the type of the cipher

u32 mask specifies the mask for the cipher

Description

Allocate a cipher handle for a random number generator. The returned struct crypto_rng is the cipher handle that is required for any subsequent API invocation for that random number generator.

For all random number generators, this call creates a new private copy of the random number generator that does not share a state with other instances. The only exception is the "krng" random number generator which is a kernel crypto API use case for the `get_random_bytes()` function of the /dev/random driver.

Return

allocated cipher handle in case of success; `IS_ERR()` is true in case of an error, PTR_ERR() returns the error code.

struct *rng_alg* * **`crypto_rng_alg`**(struct crypto_rng * *tfm*)
obtain name of RNG

Parameters

`struct crypto_rng * tfm` cipher handle

Description

Return the generic name (cra_name) of the initialized random number generator

Return

generic name string

void **`crypto_free_rng`**(struct crypto_rng * *tfm*)
zeroize and free RNG handle

Parameters

`struct crypto_rng * tfm` cipher handle to be freed

int **`crypto_rng_generate`**(struct crypto_rng * *tfm*, const u8 * *src*, unsigned int *slen*, u8 * *dst*, unsigned int *dlen*)
get random number

Parameters

`struct crypto_rng * tfm` cipher handle

`const u8 * src` Input buffer holding additional data, may be NULL

`unsigned int slen` Length of additional data

`u8 * dst` output buffer holding the random numbers

`unsigned int dlen` length of the output buffer

Description

This function fills the caller-allocated buffer with random numbers using the random number generator referenced by the cipher handle.

Return

0 function was successful; < 0 if an error occurred

int **`crypto_rng_get_bytes`**(struct crypto_rng * *tfm*, u8 * *rdata*, unsigned int *dlen*)
get random number

Parameters

`struct crypto_rng * tfm` cipher handle

`u8 * rdata` output buffer holding the random numbers

`unsigned int dlen` length of the output buffer

Description

This function fills the caller-allocated buffer with random numbers using the random number generator referenced by the cipher handle.

Return

0 function was successful; < 0 if an error occurred

int **`crypto_rng_reset`**(struct crypto_rng * *tfm*, const u8 * *seed*, unsigned int *slen*)
re-initialize the RNG

Parameters

`struct crypto_rng * tfm` cipher handle

`const u8 * seed` seed input data

`unsigned int slen` length of the seed input data

Description

The reset function completely re-initializes the random number generator referenced by the cipher handle by clearing the current state. The new state is initialized with the caller provided seed or automatically, depending on the random number generator type (the ANSI X9.31 RNG requires caller-provided seed, the SP800-90A DRBGs perform an automatic seeding). The seed is provided as a parameter to this function call. The provided seed should have the length of the seed size defined for the random number generator as defined by crypto_rng_seedsize.

Return

0 if the setting of the key was successful; < 0 if an error occurred

int **`crypto_rng_seedsize`**(struct crypto_rng * *tfm*)
obtain seed size of RNG

Parameters

`struct crypto_rng * tfm` cipher handle

Description

The function returns the seed size for the random number generator referenced by the cipher handle. This value may be zero if the random number generator does not implement or require a reseeding. For example, the SP800-90A DRBGs implement an automated reseeding after reaching a pre-defined threshold.

Return

seed size for the random number generator

Asymmetric Cipher Algorithm Definitions

struct **`akcipher_request`**
public key request

Definition

```
struct akcipher_request {
  struct crypto_async_request base;
  struct scatterlist *src;
  struct scatterlist *dst;
  unsigned int src_len;
  unsigned int dst_len;
  void *__ctx[];
};
```

Members

`base` Common attributes for async crypto requests

`src` Source data

`dst` Destination data

`src_len` Size of the input buffer

`dst_len` Size of the output buffer. It needs to be at least as big as the expected result depending on the operation After operation it will be updated with the actual size of the result. In case of error where the dst sgl size was insufficient, it will be updated to the size required for the operation.

__ctx Start of private context data

struct **akcipher_alg**
generic public key algorithm

Definition

```
struct akcipher_alg {
  int (*sign)(struct akcipher_request *req);
  int (*verify)(struct akcipher_request *req);
  int (*encrypt)(struct akcipher_request *req);
  int (*decrypt)(struct akcipher_request *req);
  int (*set_pub_key)(struct crypto_akcipher *tfm, const void *key, unsigned int keylen);
  int (*set_priv_key)(struct crypto_akcipher *tfm, const void *key, unsigned int keylen);
  unsigned int (*max_size)(struct crypto_akcipher *tfm);
  int (*init)(struct crypto_akcipher *tfm);
  void (*exit)(struct crypto_akcipher *tfm);
  unsigned int reqsize;
  struct crypto_alg base;
};
```

Members

sign Function performs a sign operation as defined by public key algorithm. In case of error, where the dst_len was insufficient, the req->dst_len will be updated to the size required for the operation

verify Function performs a sign operation as defined by public key algorithm. In case of error, where the dst_len was insufficient, the req->dst_len will be updated to the size required for the operation

encrypt Function performs an encrypt operation as defined by public key algorithm. In case of error, where the dst_len was insufficient, the req->dst_len will be updated to the size required for the operation

decrypt Function performs a decrypt operation as defined by public key algorithm. In case of error, where the dst_len was insufficient, the req->dst_len will be updated to the size required for the operation

set_pub_key Function invokes the algorithm specific set public key function, which knows how to decode and interpret the BER encoded public key

set_priv_key Function invokes the algorithm specific set private key function, which knows how to decode and interpret the BER encoded private key

max_size Function returns dest buffer size required for a given key.

init Initialize the cryptographic transformation object. This function is used to initialize the cryptographic transformation object. This function is called only once at the instantiation time, right after the transformation context was allocated. In case the cryptographic hardware has some special requirements which need to be handled by software, this function shall check for the precise requirement of the transformation and put any software fallbacks in place.

exit Deinitialize the cryptographic transformation object. This is a counterpart to **init**, used to remove various changes set in **init**.

reqsize Request context size required by algorithm implementation

base Common crypto API algorithm data structure

Asymmetric Cipher API

The Public Key API is used with the algorithms of type CRYPTO_ALG_TYPE_AKCIPHER (listed as type "akcipher" in /proc/crypto)

struct crypto_akcipher * **crypto_alloc_akcipher**(const char * *alg_name*, u32 *type*, u32 *mask*)
allocate AKCIPHER tfm handle

Parameters

`const char * alg_name` is the cra_name / name or cra_driver_name / driver name of the public key algorithm e.g. "rsa"

`u32 type` specifies the type of the algorithm

`u32 mask` specifies the mask for the algorithm

Description

Allocate a handle for public key algorithm. The returned struct crypto_akcipher is the handle that is required for any subsequent API invocation for the public key operations.

Return

allocated handle in case of success; `IS_ERR()` is true in case of an error, PTR_ERR() returns the error code.

void **`crypto_free_akcipher`**(struct crypto_akcipher * *tfm*)
free AKCIPHER tfm handle

Parameters

`struct crypto_akcipher * tfm` AKCIPHER tfm handle allocated with *`crypto_alloc_akcipher()`*

unsigned int **`crypto_akcipher_maxsize`**(struct crypto_akcipher * *tfm*)
Get len for output buffer

Parameters

`struct crypto_akcipher * tfm` AKCIPHER tfm handle allocated with *`crypto_alloc_akcipher()`*

Description

Function returns the dest buffer size required for a given key. Function assumes that the key is already set in the transformation. If this function is called without a setkey or with a failed setkey, you will end up in a NULL dereference.

int **`crypto_akcipher_encrypt`**(struct *akcipher_request* * *req*)
Invoke public key encrypt operation

Parameters

`struct akcipher_request * req` asymmetric key request

Description

Function invokes the specific public key encrypt operation for a given public key algorithm

Return

zero on success; error code in case of error

int **`crypto_akcipher_decrypt`**(struct *akcipher_request* * *req*)
Invoke public key decrypt operation

Parameters

`struct akcipher_request * req` asymmetric key request

Description

Function invokes the specific public key decrypt operation for a given public key algorithm

Return

zero on success; error code in case of error

int **`crypto_akcipher_sign`**(struct *akcipher_request* * *req*)
Invoke public key sign operation

Parameters

struct akcipher_request * req asymmetric key request

Description

Function invokes the specific public key sign operation for a given public key algorithm

Return

zero on success; error code in case of error

int **crypto_akcipher_verify**(struct *akcipher_request* * *req*)
Invoke public key verify operation

Parameters

struct akcipher_request * req asymmetric key request

Description

Function invokes the specific public key verify operation for a given public key algorithm

Return

zero on success; error code in case of error

int **crypto_akcipher_set_pub_key**(struct crypto_akcipher * *tfm*, const void * *key*, unsigned int *keylen*)
Invoke set public key operation

Parameters

struct crypto_akcipher * tfm tfm handle

const void * key BER encoded public key

unsigned int keylen length of the key

Description

Function invokes the algorithm specific set key function, which knows how to decode and interpret the encoded key

Return

zero on success; error code in case of error

int **crypto_akcipher_set_priv_key**(struct crypto_akcipher * *tfm*, const void * *key*, unsigned int *keylen*)
Invoke set private key operation

Parameters

struct crypto_akcipher * tfm tfm handle

const void * key BER encoded private key

unsigned int keylen length of the key

Description

Function invokes the algorithm specific set key function, which knows how to decode and interpret the encoded key

Return

zero on success; error code in case of error

Asymmetric Cipher Request Handle

struct *akcipher_request* * **akcipher_request_alloc**(struct crypto_akcipher * *tfm*, gfp_t *gfp*)
allocates public key request

Parameters

`struct crypto_akcipher * tfm` AKCIPHER tfm handle allocated with *`crypto_alloc_akcipher()`*

`gfp_t gfp` allocation flags

Return

allocated handle in case of success or NULL in case of an error.

void **`akcipher_request_free`**(struct *akcipher_request* * *req*)
zeroize and free public key request

Parameters

`struct akcipher_request * req` request to free

void **`akcipher_request_set_callback`**(struct *akcipher_request* * *req*, u32 *flgs*, crypto_completion_t *cmpl*, void * *data*)
Sets an asynchronous callback.

Parameters

`struct akcipher_request * req` request that the callback will be set for

`u32 flgs` specify for instance if the operation may backlog

`crypto_completion_t cmpl` callback which will be called

`void * data` private data used by the caller

Description

Callback will be called when an asynchronous operation on a given request is finished.

void **`akcipher_request_set_crypt`**(struct *akcipher_request* * *req*, struct scatterlist * *src*, struct scatterlist * *dst*, unsigned int *src_len*, unsigned int *dst_len*)
Sets request parameters

Parameters

`struct akcipher_request * req` public key request

`struct scatterlist * src` ptr to input scatter list

`struct scatterlist * dst` ptr to output scatter list

`unsigned int src_len` size of the src input scatter list to be processed

`unsigned int dst_len` size of the dst output scatter list

Description

Sets parameters required by crypto operation

Key-agreement Protocol Primitives (KPP) Cipher Algorithm Definitions

struct **`kpp_request`**

Definition

```
struct kpp_request {
  struct crypto_async_request base;
  struct scatterlist *src;
  struct scatterlist *dst;
  unsigned int src_len;
  unsigned int dst_len;
  void *__ctx[];
};
```

Members

`base` Common attributes for async crypto requests

`src` Source data

`dst` Destination data

`src_len` Size of the input buffer

`dst_len` Size of the output buffer. It needs to be at least as big as the expected result depending on the operation After operation it will be updated with the actual size of the result. In case of error where the dst sgl size was insufficient, it will be updated to the size required for the operation.

`__ctx` Start of private context data

struct **`crypto_kpp`**
user-instantiated object which encapsulate algorithms and core processing logic

Definition

```
struct crypto_kpp {
  struct crypto_tfm base;
};
```

Members

`base` Common crypto API algorithm data structure

struct **`kpp_alg`**
generic key-agreement protocol primitives

Definition

```
struct kpp_alg {
  int (*set_secret)(struct crypto_kpp *tfm, const void *buffer, unsigned int len);
  int (*generate_public_key)(struct kpp_request *req);
  int (*compute_shared_secret)(struct kpp_request *req);
  unsigned int (*max_size)(struct crypto_kpp *tfm);
  int (*init)(struct crypto_kpp *tfm);
  void (*exit)(struct crypto_kpp *tfm);
  unsigned int reqsize;
  struct crypto_alg base;
};
```

Members

`set_secret` Function invokes the protocol specific function to store the secret private key along with parameters. The implementation knows how to decode the buffer

`generate_public_key` Function generate the public key to be sent to the counterpart. In case of error, where output is not big enough req->dst_len will be updated to the size required

`compute_shared_secret` Function compute the shared secret as defined by the algorithm. The result is given back to the user. In case of error, where output is not big enough, req->dst_len will be updated to the size required

`max_size` Function returns the size of the output buffer

`init` Initialize the object. This is called only once at instantiation time. In case the cryptographic hardware needs to be initialized. Software fallback should be put in place here.

`exit` Undo everything **init** did.

`reqsize` Request context size required by algorithm implementation

base Common crypto API algorithm data structure

struct **kpp_secret**
small header for packing secret buffer

Definition

```
struct kpp_secret {
  unsigned short type;
  unsigned short len;
};
```

Members

type define type of secret. Each kpp type will define its own

len specify the len of the secret, include the header, that follows the struct

Key-agreement Protocol Primitives (KPP) Cipher API

The KPP API is used with the algorithm type CRYPTO_ALG_TYPE_KPP (listed as type "kpp" in /proc/crypto)

struct *crypto_kpp* * **crypto_alloc_kpp**(const char * *alg_name*, u32 *type*, u32 *mask*)
allocate KPP tfm handle

Parameters

const char * alg_name is the name of the kpp algorithm (e.g. "dh", "ecdh")

u32 type specifies the type of the algorithm

u32 mask specifies the mask for the algorithm

Description

Allocate a handle for kpp algorithm. The returned struct crypto_kpp is required for any following API invocation

Return

allocated handle in case of success; IS_ERR() is true in case of an error, PTR_ERR() returns the error code.

void **crypto_free_kpp**(struct *crypto_kpp* * *tfm*)
free KPP tfm handle

Parameters

struct crypto_kpp * tfm KPP tfm handle allocated with *crypto_alloc_kpp()*

int **crypto_kpp_set_secret**(struct *crypto_kpp* * *tfm*, const void * *buffer*, unsigned int *len*)
Invoke kpp operation

Parameters

struct crypto_kpp * tfm tfm handle

const void * buffer Buffer holding the packet representation of the private key. The structure of the packet key depends on the particular KPP implementation. Packing and unpacking helpers are provided for ECDH and DH (see the respective header files for those implementations).

unsigned int len Length of the packet private key buffer.

Description

Function invokes the specific kpp operation for a given alg.

Return

zero on success; error code in case of error

int **crypto_kpp_generate_public_key**(struct *kpp_request* * *req*)
Invoke kpp operation

Parameters

struct kpp_request * req kpp key request

Description

Function invokes the specific kpp operation for generating the public part for a given kpp algorithm.

To generate a private key, the caller should use a random number generator. The output of the requested length serves as the private key.

Return

zero on success; error code in case of error

int **crypto_kpp_compute_shared_secret**(struct *kpp_request* * *req*)
Invoke kpp operation

Parameters

struct kpp_request * req kpp key request

Description

Function invokes the specific kpp operation for computing the shared secret for a given kpp algorithm.

Return

zero on success; error code in case of error

unsigned int **crypto_kpp_maxsize**(struct *crypto_kpp* * *tfm*)
Get len for output buffer

Parameters

struct crypto_kpp * tfm KPP tfm handle allocated with *crypto_alloc_kpp()*

Description

Function returns the output buffer size required for a given key. Function assumes that the key is already set in the transformation. If this function is called without a setkey or with a failed setkey, you will end up in a NULL dereference.

Key-agreement Protocol Primitives (KPP) Cipher Request Handle

struct *kpp_request* * **kpp_request_alloc**(struct *crypto_kpp* * *tfm*, gfp_t *gfp*)
allocates kpp request

Parameters

struct crypto_kpp * tfm KPP tfm handle allocated with *crypto_alloc_kpp()*

gfp_t gfp allocation flags

Return

allocated handle in case of success or NULL in case of an error.

void **kpp_request_free**(struct *kpp_request* * *req*)
zeroize and free kpp request

Parameters

struct kpp_request * req request to free

void **kpp_request_set_callback**(struct *kpp_request* * *req*, u32 *flgs*, crypto_completion_t *cmpl*, void * *data*)
Sets an asynchronous callback.

Parameters

struct kpp_request * req request that the callback will be set for

u32 flgs specify for instance if the operation may backlog

crypto_completion_t cmpl callback which will be called

void * data private data used by the caller

Description

Callback will be called when an asynchronous operation on a given request is finished.

void **kpp_request_set_input**(struct *kpp_request* * *req*, struct scatterlist * *input*, unsigned int *input_len*)

Sets input buffer

Parameters

struct kpp_request * req kpp request

struct scatterlist * input ptr to input scatter list

unsigned int input_len size of the input scatter list

Description

Sets parameters required by generate_public_key

void **kpp_request_set_output**(struct *kpp_request* * *req*, struct scatterlist * *output*, unsigned int *output_len*)

Sets output buffer

Parameters

struct kpp_request * req kpp request

struct scatterlist * output ptr to output scatter list

unsigned int output_len size of the output scatter list

Description

Sets parameters required by kpp operation

ECDH Helper Functions

To use ECDH with the KPP cipher API, the following data structure and functions should be used.

The ECC curves known to the ECDH implementation are specified in this header file.

To use ECDH with KPP, the following functions should be used to operate on an ECDH private key. The packet private key that can be set with the KPP API function call of crypto_kpp_set_secret.

struct **ecdh**

define an ECDH private key

Definition

```
struct ecdh {
  unsigned short curve_id;
  char *key;
  unsigned short key_size;
};
```

Members

curve_id ECC curve the key is based on.

key Private ECDH key

key_size Size of the private ECDH key

unsigned int **crypto_ecdh_key_len**(const struct *ecdh* * *params*)
Obtain the size of the private ECDH key

Parameters

const struct ecdh * params private ECDH key

Description

This function returns the packet ECDH key size. A caller can use that with the provided ECDH private key reference to obtain the required memory size to hold a packet key.

Return

size of the key in bytes

int **crypto_ecdh_encode_key**(char * *buf*, unsigned int *len*, const struct *ecdh* * *p*)
encode the private key

Parameters

char * buf Buffer allocated by the caller to hold the packet ECDH private key. The buffer should be at least crypto_ecdh_key_len bytes in size.

unsigned int len Length of the packet private key buffer

const struct ecdh * p Buffer with the caller-specified private key

Description

The ECDH implementations operate on a packet representation of the private key.

Return

-EINVAL if buffer has insufficient size, 0 on success

int **crypto_ecdh_decode_key**(const char * *buf*, unsigned int *len*, struct *ecdh* * *p*)
decode a private key

Parameters

const char * buf Buffer holding a packet key that should be decoded

unsigned int len Length of the packet private key buffer

struct ecdh * p Buffer allocated by the caller that is filled with the unpacked ECDH private key.

Description

The unpacking obtains the private key by pointing **p** to the correct location in **buf**. Thus, both pointers refer to the same memory.

Return

-EINVAL if buffer has insufficient size, 0 on success

DH Helper Functions

To use DH with the KPP cipher API, the following data structure and functions should be used.

To use DH with KPP, the following functions should be used to operate on a DH private key. The packet private key that can be set with the KPP API function call of crypto_kpp_set_secret.

struct **dh**
define a DH private key

Definition

```
struct dh {
  void *key;
  void *p;
  void *g;
  unsigned int key_size;
  unsigned int p_size;
  unsigned int g_size;
};
```

Members

key Private DH key

p Diffie-Hellman parameter P

g Diffie-Hellman generator G

key_size Size of the private DH key

p_size Size of DH parameter P

g_size Size of DH generator G

unsigned int **crypto_dh_key_len**(const struct *dh* * *params*)
Obtain the size of the private DH key

Parameters

const struct dh * params private DH key

Description

This function returns the packet DH key size. A caller can use that with the provided DH private key reference to obtain the required memory size to hold a packet key.

Return

size of the key in bytes

int **crypto_dh_encode_key**(char * *buf*, unsigned int *len*, const struct *dh* * *params*)
encode the private key

Parameters

char * buf Buffer allocated by the caller to hold the packet DH private key. The buffer should be at least crypto_dh_key_len bytes in size.

unsigned int len Length of the packet private key buffer

const struct dh * params Buffer with the caller-specified private key

Description

The DH implementations operate on a packet representation of the private key.

Return

-EINVAL if buffer has insufficient size, 0 on success

int **crypto_dh_decode_key**(const char * *buf*, unsigned int *len*, struct *dh* * *params*)
decode a private key

Parameters

const char * buf Buffer holding a packet key that should be decoded

unsigned int len Length of the packet private key buffer

struct dh * params Buffer allocated by the caller that is filled with the unpacked DH private key.

Description

The unpacking obtains the private key by pointing **p** to the correct location in **buf**. Thus, both pointers refer to the same memory.

Return

-EINVAL if buffer has insufficient size, 0 on success

CHAPTER

SEVEN

CODE EXAMPLES

Code Example For Symmetric Key Cipher Operation

```
/* tie all data structures together */
struct skcipher_def {
    struct scatterlist sg;
    struct crypto_skcipher *tfm;
    struct skcipher_request *req;
    struct crypto_wait wait;
};

/* Perform cipher operation */
static unsigned int test_skcipher_encdec(struct skcipher_def *sk,
                     int enc)
{
    int rc;

    if (enc)
        rc = crypto_wait_req(crypto_skcipher_encrypt(sk->req), &sk->wait);
    else
        rc = crypto_wait_req(crypto_skcipher_decrypt(sk->req), &sk->wait);

    if (rc)
            pr_info("skcipher encrypt returned with result %d\n", rc);

    return rc;
}

/* Initialize and trigger cipher operation */
static int test_skcipher(void)
{
    struct skcipher_def sk;
    struct crypto_skcipher *skcipher = NULL;
    struct skcipher_request *req = NULL;
    char *scratchpad = NULL;
    char *ivdata = NULL;
    unsigned char key[32];
    int ret = -EFAULT;

    skcipher = crypto_alloc_skcipher("cbc-aes-aesni", 0, 0);
    if (IS_ERR(skcipher)) {
        pr_info("could not allocate skcipher handle\n");
        return PTR_ERR(skcipher);
    }

    req = skcipher_request_alloc(skcipher, GFP_KERNEL);
    if (!req) {
        pr_info("could not allocate skcipher request\n");
        ret = -ENOMEM;
        goto out;
    }

    skcipher_request_set_callback(req, CRYPTO_TFM_REQ_MAY_BACKLOG,
                      crypto_req_done,
                      &sk.wait);

    /* AES 256 with random key */
    get_random_bytes(&key, 32);
    if (crypto_skcipher_setkey(skcipher, key, 32)) {
        pr_info("key could not be set\n");
        ret = -EAGAIN;
        goto out;
    }

    /* IV will be random */
    ivdata = kmalloc(16, GFP_KERNEL);
    if (!ivdata) {
        pr_info("could not allocate ivdata\n");
        goto out;
    }
    get_random_bytes(ivdata, 16);

    /* Input data will be random */
    scratchpad = kmalloc(16, GFP_KERNEL);
    if (!scratchpad) {
        pr_info("could not allocate scratchpad\n");
        goto out;
    }
    get_random_bytes(scratchpad, 16);

    sk.tfm = skcipher;
    sk.req = req;

    /* We encrypt one block */
    sg_init_one(&sk.sg, scratchpad, 16);
    skcipher_request_set_crypt(req, &sk.sg, &sk.sg, 16, ivdata);
    crypto_init_wait(&sk.wait);

    /* encrypt data */
    ret = test_skcipher_encdec(&sk, 1);
    if (ret)
        goto out;

    pr_info("Encryption triggered successfully\n");

out:
    if (skcipher)
        crypto_free_skcipher(skcipher);
    if (req)
        skcipher_request_free(req);
    if (ivdata)
        kfree(ivdata);
    if (scratchpad)
        kfree(scratchpad);
    return ret;
}
```

Code Example For Use of Operational State Memory With SHASH

```
struct sdesc {
    struct shash_desc shash;
    char ctx[];
};

static struct sdesc *init_sdesc(struct crypto_shash *alg)
{
    struct sdesc *sdesc;
    int size;

    size = sizeof(struct shash_desc) + crypto_shash_descsize(alg);
    sdesc = kmalloc(size, GFP_KERNEL);
    if (!sdesc)
        return ERR_PTR(-ENOMEM);
    sdesc->shash.tfm = alg;
    sdesc->shash.flags = 0x0;
    return sdesc;
}

static int calc_hash(struct crypto_shash *alg,
             const unsigned char *data, unsigned int datalen,
             unsigned char *digest)
{
    struct sdesc *sdesc;
    int ret;

    sdesc = init_sdesc(alg);
    if (IS_ERR(sdesc)) {
        pr_info("can't alloc sdesc\n");
        return PTR_ERR(sdesc);
    }

    ret = crypto_shash_digest(&sdesc->shash, data, datalen, digest);
    kfree(sdesc);
    return ret;
}

static int test_hash(const unsigned char *data, unsigned int datalen,
             unsigned char *digest)
{
    struct crypto_shash *alg;
    char *hash_alg_name = "sha1-padlock-nano";
    int ret;

    alg = crypto_alloc_shash(hash_alg_name, CRYPTO_ALG_TYPE_SHASH, 0);
    if (IS_ERR(alg)) {
            pr_info("can't alloc alg %s\n", hash_alg_name);
            return PTR_ERR(alg);
    }
    ret = calc_hash(alg, data, datalen, digest);
    crypto_free_shash(alg);
    return ret;
}
```

Code Example For Random Number Generator Usage

```
static int get_random_numbers(u8 *buf, unsigned int len)
{
    struct crypto_rng *rng = NULL;
    char *drbg = "drbg_nopr_sha256"; /* Hash DRBG with SHA-256, no PR */
    int ret;

    if (!buf || !len) {
        pr_debug("No output buffer provided\n");
        return -EINVAL;
    }

    rng = crypto_alloc_rng(drbg, 0, 0);
    if (IS_ERR(rng)) {
        pr_debug("could not allocate RNG handle for %s\n", drbg);
        return PTR_ERR(rng);
    }

    ret = crypto_rng_get_bytes(rng, buf, len);
    if (ret < 0)
        pr_debug("generation of random numbers failed\n");
    else if (ret == 0)
        pr_debug("RNG returned no data");
    else
        pr_debug("RNG returned %d bytes of data\n", ret);

out:
    crypto_free_rng(rng);
    return ret;
}
```