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
    - 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 generator (obsolete)

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_KPP Key-agreement Protocol Primitive (KPP) such as an ECDH or DH implementation
- CRYPTO_ALG_TYPE_HASH Raw message 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.

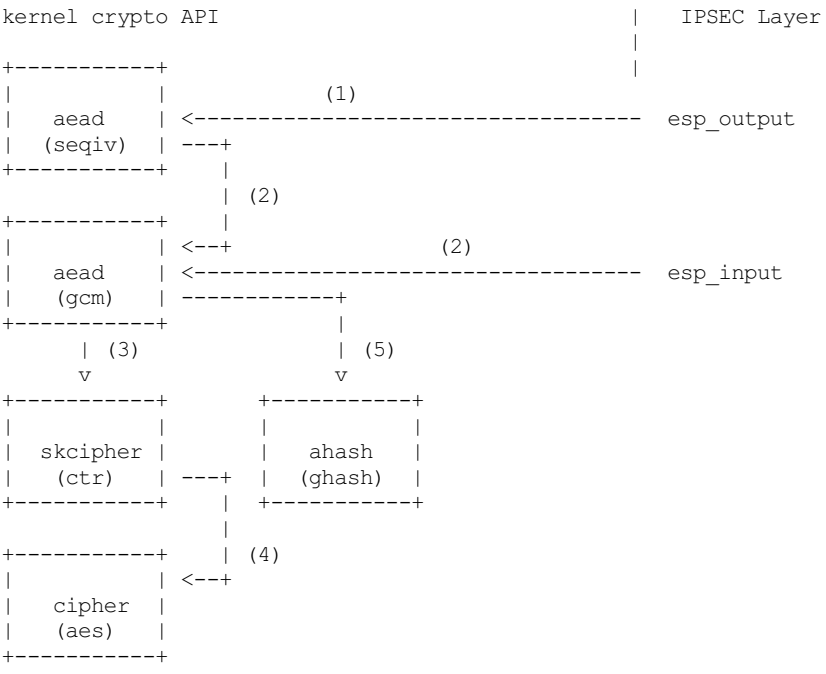
The depicted example decomposes the AEAD cipher of GCM(AES) based on the generic C implementations (gcm.c, aes-generic.c, ctr.c, ghash-generic.c, seqiv.c). The generic implementation serves as an example showing the complete logic of the kernel crypto API.

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.



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 seqiv(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.

   The SEQIV generates the IV.

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

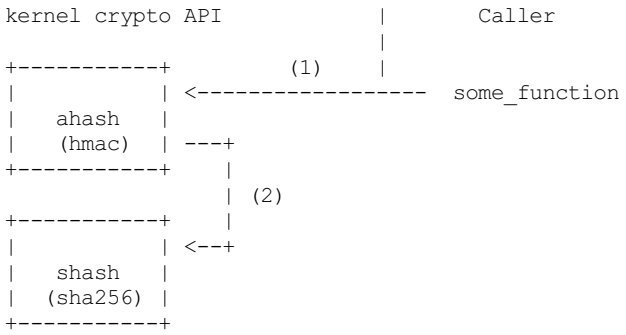
Generic block ciphers follow the same concept as depicted with the ASCII art picture above.

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:



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.