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.

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

The registration functions return 0 on success, or a negative errno value on failure. crypto_register_algs() succeeds only if it successfully registered all the given algorithms; if it fails partway through, then any changes are rolled back.

The unregistration functions always succeed, so they don't have a return value. Don't try to unregister algorithms that aren't currently registered.

Single-Block Symmetric Ciphers [CIPHER]

Example of transformations: aes, serpent, ...

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), chacha20, ...

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 skcipher_alg

Struct skcipher_alg defines a multi-block cipher, or more generally, a length-preserving symmetric cipher algorithm.

Scatterlist handling

Some 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:

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

void crypto_unregister_shash(struct shash_alg *alg);
void 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.