Inline Encryption

Background

Inline encryption hardware sits logically between memory and disk, and can en/decrypt data as it goes in/out of the disk. For each I/O request, software can control exactly how the inline encryption hardware will en/decrypt the data in terms of key, algorithm, data unit size (the granularity of en/decryption), and data unit number (a value that determines the initialization vector(s)).

Some inline encryption hardware accepts all encryption parameters including raw keys directly in low-level I/O requests. However, most inline encryption hardware instead has a fixed number of "keyslots" and requires that the key, algorithm, and data unit size first be programmed into a keyslot. Each low-level I/O request then just contains a keyslot index and data unit number.

Note that inline encryption hardware is very different from traditional crypto accelerators, which are supported through the kernel crypto API. Traditional crypto accelerators operate on memory regions, whereas inline encryption hardware operates on I/O requests. Thus, inline encryption hardware needs to be managed by the block layer, not the kernel crypto API.

Inline encryption hardware is also very different from "self-encrypting drives", such as those based on the TCG Opal or ATA Security standards. Self-encrypting drives don't provide fine-grained control of encryption and provide no way to verify the correctness of the resulting ciphertext. Inline encryption hardware provides fine-grained control of encryption, including the choice of key and initialization vector for each sector, and can be tested for correctness.

Objective

We want to support inline encryption in the kernel. To make testing easier, we also want support for falling back to the kernel crypto API when actual inline encryption hardware is absent. We also want inline encryption to work with layered devices like device-mapper and loopback (i.e. we want to be able to use the inline encryption hardware of the underlying devices if present, or else fall back to crypto API en/decryption).

Constraints and notes

- We need a way for upper layers (e.g. filesystems) to specify an encryption context to use for en/decrypting a bio, and device drivers (e.g. UFSHCD) need to be able to use that encryption context when they process the request. Encryption contexts also introduce constraints on bio merging; the block layer needs to be aware of these constraints.
- Different inline encryption hardware has different supported algorithms, supported data unit sizes, maximum data unit numbers, etc. We call these properties the "crypto capabilities". We need a way for device drivers to advertise crypto capabilities to upper layers in a generic way.
- Inline encryption hardware usually (but not always) requires that keys be programmed into keyslots before being used. Since programming keyslots may be slow and there may not be very many keyslots, we shouldn't just program the key for every I/O request, but rather keep track of which keys are in the keyslots and reuse an already-programmed keyslot when possible.
- Upper layers typically define a specific end-of-life for crypto keys, e.g. when an encrypted directory is locked or when a crypto mapping is torn down. At these times, keys are wiped from memory. We must provide a way for upper layers to also evict keys from any keyslots they are present in.
- When possible, device-mapper devices must be able to pass through the inline encryption support of their underlying devices. However, it doesn't make sense for device-mapper devices to have keyslots themselves.

Basic design

We introduce `struct blk_crypto_key` to represent an inline encryption key and how it will be used. This includes the actual bytes of the key; the size of the key; the algorithm and data unit size the key will be used with; and the number of bytes needed to represent the maximum data unit number the key will be used with.

We introduce `struct bio_crypt_ctx` to represent an encryption context. It contains a data unit number and a pointer to a blk_crypto_key. We add pointers to a bio_crypt_ctx to `struct bio` and `struct request`; this allows users of the block layer (e.g. filesystems) to provide an encryption context when creating a bio and have it be passed down the stack for processing by the block layer and device drivers. Note that the encryption context doesn't explicitly say whether to encrypt or decrypt, as that is implicit from the direction of the bio; WRITE means encrypt, and READ means decrypt.

We also introduce `struct blk_crypto_profile` to contain all generic inline encryption-related state for a particular inline encryption device. The blk_crypto_profile serves as the way that drivers for inline encryption hardware advertise their crypto capabilities and provide certain functions (e.g., functions to program and evict keys) to upper layers. Each device driver that wants to support inline encryption will construct a blk_crypto_profile, then associate it with the disk's request_queue.

The blk_crypto_profile also manages the hardware's keyslots, when applicable. This happens in the block layer, so that users of the block layer can just specify encryption contexts and don't need to know about keyslots at all, nor do device drivers need to care about most details of keyslot management.

Specifically, for each keyslot, the block layer (via the blk_crypto_profile) keeps track of which blk_crypto_key that keyslot contains (if any), and how many in-flight I/O requests are using it. When the block layer creates a `struct request` for a bio that has an encryption context, it grabs a keyslot that already contains the key if possible. Otherwise it waits for an idle keyslot (a keyslot that isn't in-use by any I/O), then programs the key into the least-recently-used idle keyslot using the function the device driver provided. In both cases, the resulting keyslot is stored in the `crypt_keyslot` field of the request, where it is then accessible to device drivers and is released after the request completes.

`struct request` also contains a pointer to the original bio_crypt_ctx. Requests can be built from multiple bios, and the block layer must take the encryption context into account when trying to merge bios and requests. For two bios/requests to be merged, they must have compatible encryption contexts: both unencrypted, or both encrypted with the same key and contiguous data unit numbers. Only the encryption context for the first bio in a request is retained, since the remaining bios have been verified to be merge-compatible with the first bio.

To make it possible for inline encryption to work with request_queue based layered devices, when a request is cloned, its encryption context is cloned as well. When the cloned request is submitted, it is then processed as usual; this includes getting a keyslot from the clone's target device if needed.

blk-crypto-fallback

It is desirable for the inline encryption support of upper layers (e.g. filesystems) to be testable without real inline encryption hardware, and likewise for the block layer's keyslot management logic. It is also desirable to allow upper layers to just always use inline encryption rather than have to implement encryption in multiple ways.

Therefore, we also introduce *blk-crypto-fallback*, which is an implementation of inline encryption using the kernel crypto API. blk-crypto-fallback is built into the block layer, so it works on any block device without any special setup. Essentially, when a bio with an encryption context is submitted to a request_queue that doesn't support that encryption context, the block layer will handle en/decryption of the bio using blk-crypto-fallback.

For encryption, the data cannot be encrypted in-place, as callers usually rely on it being unmodified. Instead, blk-crypto-fallback allocates bounce pages, fills a new bio with those bounce pages, encrypts the data into those bounce pages, and submits that "bounce" bio. When the bounce bio completes, blk-crypto-fallback completes the original bio. If the original bio is too large, multiple bounce bios may be required; see the code for details.

For decryption, blk-crypto-fallback "wraps" the bio's completion callback (`bi_complete`) and private data (`bi_private`) with its own, unsets the bio's encryption context, then submits the bio. If the read completes successfully, blk-crypto-fallback restores the bio's original completion callback and private data, then decrypts the bio's data in-place using the kernel crypto API. Decryption happens from a workqueue, as it may sleep. Afterwards, blk-crypto-fallback completes the bio.

In both cases, the bios that blk-crypto-fallback submits no longer have an encryption context. Therefore, lower layers only see standard unencrypted I/O.

blk-crypto-fallback also defines its own blk_crypto_profile and has its own "keyslots"; its keyslots contain `struct crypto_skcipher` objects. The reason for this is twofold. First, it allows the keyslot management logic to be tested without actual inline encryption hardware. Second, similar to actual inline encryption hardware, the crypto API doesn't accept keys directly in requests but rather requires that keys be set ahead of time, and setting keys can be expensive; moreover, allocating a crypto_skcipher can't happen on the I/O path at all due to the locks it takes. Therefore, the concept of keyslots still makes sense for blk-crypto-fallback.

Note that regardless of whether real inline encryption hardware or blk-crypto-fallback is used, the ciphertext written to disk (and hence the on-disk format of data) will be the same (assuming that both the inline encryption hardware's implementation and the kernel crypto API's implementation of the algorithm being used adhere to spec and function correctly).

blk-crypto-fallback is optional and is controlled by the `CONFIG_BLK_INLINE_ENCRYPTION_FALLBACK` kernel configuration option.

API presented to users of the block layer

`blk_crypto_config_supported()` allows users to check ahead of time whether inline encryption with particular crypto settings will work on a particular request_queue -- either via hardware or via blk-crypto-fallback. This function takes in a `struct blk_crypto_config` which is like blk_crypto_key, but omits the actual bytes of the key and instead just contains the algorithm, data unit size, etc. This function can be useful if blk-crypto-fallback is disabled.

`blk_crypto_init_key()` allows users to initialize a blk_crypto_key.

Users must call `blk_crypto_start_using_key()` before actually starting to use a blk_crypto_key on a request_queue (even if `blk_crypto_config_supported()` was called earlier). This is needed to initialize blk-crypto-fallback if it will be needed. This must not be called from the data path, as this may have to allocate resources, which may deadlock in that case.

Next, to attach an encryption context to a bio, users should call `bio_crypt_set_ctx()`. This function allocates a bio_crypt_ctx and attaches it to a bio, given the blk_crypto_key and the data unit number that will be used for en/decryption. Users don't need to worry about freeing the bio_crypt_ctx later, as that happens automatically when the bio is freed or reset.

Finally, when done using inline encryption with a blk_crypto_key on a request_queue, users must call `blk_crypto_evict_key()`. This ensures that the key is evicted from all keyslots it may be programmed into and unlinked from any kernel data structures it may be linked into.

In summary, for users of the block layer, the lifecycle of a blk_crypto_key is as follows:

1. `blk_crypto_config_supported()` (optional)
2. `blk_crypto_init_key()`
3. `blk_crypto_start_using_key()`
4. `bio_crypt_set_ctx()` (potentially many times)
5. `blk_crypto_evict_key()` (after all I/O has completed)
6. Zeroize the blk_crypto_key (this has no dedicated function)

If a blk_crypto_key is being used on multiple request_queues, then `blk_crypto_config_supported()` (if used), `blk_crypto_start_using_key()`, and `blk_crypto_evict_key()` must be called on each request_queue.

API presented to device drivers

A device driver that wants to support inline encryption must set up a blk_crypto_profile in the request_queue of its device. To do this, it first must call `blk_crypto_profile_init()` (or its resource-managed variant `devm_blk_crypto_profile_init()`), providing the number of keyslots.

Next, it must advertise its crypto capabilities by setting fields in the blk_crypto_profile, e.g. `modes_supported` and `max_dun_bytes_supported`.

It then must set function pointers in the `ll_ops` field of the blk_crypto_profile to tell upper layers how to control the inline encryption hardware, e.g. how to program and evict keyslots. Most drivers will need to implement `keyslot_program` and `keyslot_evict`. For details, see the comments for `struct blk_crypto_ll_ops`.

Once the driver registers a blk_crypto_profile with a request_queue, I/O requests the driver receives via that queue may have an encryption context. All encryption contexts will be compatible with the crypto capabilities declared in the blk_crypto_profile, so drivers don't need to worry about handling unsupported requests. Also, if a nonzero number of keyslots was declared in the blk_crypto_profile, then all I/O requests that have an encryption context will also have a keyslot which was already programmed with the appropriate key.

If the driver implements runtime suspend and its blk_crypto_ll_ops don't work while the device is runtime-suspended, then the driver must also set the `dev` field of the blk_crypto_profile to point to the `struct device` that will be resumed before any of the low-level operations are called.

If there are situations where the inline encryption hardware loses the contents of its keyslots, e.g. device resets, the driver must handle reprogramming the keyslots. To do this, the driver may call `blk_crypto_reprogram_all_keys()`.

Finally, if the driver used `blk_crypto_profile_init()` instead of `devm_blk_crypto_profile_init()`, then it is responsible for calling `blk_crypto_profile_destroy()` when the crypto profile is no longer needed.

Layered Devices

Request queue based layered devices like dm-rq that wish to support inline encryption need to create their own blk_crypto_profile for their request_queue, and expose whatever functionality they choose. When a layered device wants to pass a clone of that request to another request_queue, blk-crypto will initialize and prepare the clone as necessary; see `blk_crypto_insert_cloned_request()`.

Interaction between inline encryption and blk integrity

At the time of this patch, there is no real hardware that supports both these features. However, these features do interact with each other, and it's not completely trivial to make them both work together properly. In particular, when a WRITE bio wants to use inline encryption on a device that supports both features, the bio will have an encryption context specified, after which its integrity information is calculated (using the plaintext data, since the encryption will happen while data is being written), and the data and integrity info is sent to the device. Obviously, the integrity info must be verified before the data is encrypted. After the data is encrypted, the device must not store the integrity info that it received with the plaintext data since that might reveal information about the plaintext data. As such, it must re-generate the integrity info from the ciphertext data and store that on disk instead. Another issue with storing the integrity info of the plaintext data is that it changes the on disk format depending on whether hardware inline encryption support is present or the kernel crypto API fallback is used (since if the fallback is used, the device will receive the integrity info of the ciphertext, not that of the plaintext).

Because there isn't any real hardware yet, it seems prudent to assume that hardware implementations might not implement both features together correctly, and disallow the combination for now. Whenever a device supports integrity, the kernel will pretend that the device does not support hardware inline encryption (by setting the blk_crypto_profile in the request_queue of the device to NULL). When the crypto API fallback is enabled, this means that all bios with and encryption context will use the fallback, and IO will complete as usual. When the fallback is disabled, a bio with an encryption context will be failed.