Asymmetric / Public-key Cryptography Key Type

Overview

The "asymmetric" key type is designed to be a container for the keys used in public-key cryptography, without imposing any particular restrictions on the form or mechanism of the cryptography or form of the key.

The asymmetric key is given a subtype that defines what sort of data is associated with the key and provides operations to describe and destroy it. However, no requirement is made that the key data actually be stored in the key.

A completely in-kernel key retention and operation subtype can be defined, but it would also be possible to provide access to cryptographic hardware (such as a TPM) that might be used to both retain the relevant key and perform operations using that key. In such a case, the asymmetric key would then merely be an interface to the TPM driver.

Also provided is the concept of a data parser. Data parsers are responsible for extracting information from the blobs of data passed to the instantiation function. The first data parser that recognises the blob gets to set the subtype of the key and define the operations that can be done on that key.

A data parser may interpret the data blob as containing the bits representing a key, or it may interpret it as a reference to a key held somewhere else in the system (for example, a TPM).

Key Identification

If a key is added with an empty name, the instantiation data parsers are given the opportunity to pre-parse a key and to determine the description the key should be given from the content of the key.

This can then be used to refer to the key, either by complete match or by partial match. The key type may also use other criteria to refer to a key.

The asymmetric key type's match function can then perform a wider range of comparisons than just the straightforward comparison of the description with the criterion string:

1. If the criterion string is of the form "id:<hexdigits>" then the match function will examine a key's fingerprint to see if the hex digits given after the "id:" match the tail. For instance:

   ```
   keyctl search @s asymmetric id:5acc2142
   ```

   will match a key with fingerprint:

   ```
   1A00 2040 7601 7889 DE11  882C 3823 04AD 5ACC 2142
   ```

2. If the criterion string is of the form "<subtype>:<hexdigits>" then the match will match the ID as in (1), but with the added restriction that only keys of the specified subtype (e.g. tpm) will be matched. For instance:

   ```
   keyctl search @s asymmetric tpm:5acc2142
   ```

Looking in /proc/keys, the last 8 hex digits of the key fingerprint are displayed, along with the subtype:

```
1a39e171 I-----     1 perm 3f010000     0     0 asymmetric modsign.0: DSA 5acc2142 []
```

Accessing Asymmetric Keys

For general access to asymmetric keys from within the kernel, the following inclusion is required:

```
#include <crypto/public_key.h>
```

This gives access to functions for dealing with asymmetric / public keys. Three enums are defined there for representing public-key cryptography algorithms:

```
enum pkey_algo
```

digest algorithms used by those:

```
enum pkey_hash_algo
```

and key identifier representations:

```
enum pkey_id_type
```

Note that the key type representation types are required because key identifiers from different standards aren't necessarily compatible. For instance, PGP generates key identifiers by hashing the key data plus some PGP-specific metadata, whereas X.509 has arbitrary certificate identifiers.

The operations defined upon a key are:

1. Signature verification.

Other operations are possible (such as encryption) with the same key data required for verification, but not currently supported, and others (eg. decryption and signature generation) require extra key data.

Signature Verification

An operation is provided to perform cryptographic signature verification, using an asymmetric key to provide or to provide access to the public key:

```
int verify_signature(const struct key *key,
                     const struct public_key_signature *sig);
```

The caller must have already obtained the key from some source and can then use it to check the signature. The caller must have parsed the signature and transferred the relevant bits to the structure pointed to by sig:

```
struct public_key_signature {
        u8 *digest;
        u8 digest_size;
        enum pkey_hash_algo pkey_hash_algo : 8;
        u8 nr_mpi;
        union {
                MPI mpi[2];
                ...
        };
};
```

The algorithm used must be noted in sig->pkey_hash_algo, and all the MPIs that make up the actual signature must be stored in sig->mpi[] and the count of MPIs placed in sig->nr_mpi.

In addition, the data must have been digested by the caller and the resulting hash must be pointed to by sig->digest and the size of the hash be placed in sig->digest_size.

The function will return 0 upon success or -EKEYREJECTED if the signature doesn't match.

The function may also return -ENOTSUPP if an unsupported public-key algorithm or public-key/hash algorithm combination is specified or the key doesn't support the operation; -EBADMSG or -ERANGE if some of the parameters have weird data; or -ENOMEM if an allocation can't be performed. -EINVAL can be returned if the key argument is the wrong type or is incompletely set up.

Asymmetric Key Subtypes

Asymmetric keys have a subtype that defines the set of operations that can be performed on that key and that determines what data is attached as the key payload. The payload format is entirely at the whim of the subtype.

The subtype is selected by the key data parser and the parser must initialise the data required for it. The asymmetric key retains a reference on the subtype module.

The subtype definition structure can be found in:

```
#include <keys/asymmetric-subtype.h>
```

and looks like the following:

```
struct asymmetric_key_subtype {
        struct module           *owner;
        const char              *name;

        void (*describe)(const struct key *key, struct seq_file *m);
        void (*destroy)(void *payload);
        int (*query)(const struct kernel_pkey_params *params,
                     struct kernel_pkey_query *info);
        int (*eds_op)(struct kernel_pkey_params *params,
                      const void *in, void *out);
        int (*verify_signature)(const struct key *key,
                                const struct public_key_signature *sig);
};
```

Asymmetric keys point to this with their payload[asym_subtype] member.

The owner and name fields should be set to the owning module and the name of the subtype. Currently, the name is only used for print statements.

There are a number of operations defined by the subtype:

1. describe().

   Mandatory. This allows the subtype to display something in /proc/keys against the key. For instance the name of the public key algorithm type could be displayed. The key type will display the tail of the key identity string after

this.

2. destroy().

   Mandatory. This should free the memory associated with the key. The asymmetric key will look after freeing the fingerprint and releasing the reference on the subtype module.

3. query().

   Mandatory. This is a function for querying the capabilities of a key.

4. eds_op().

   Optional. This is the entry point for the encryption, decryption and signature creation operations (which are distinguished by the operation ID in the parameter struct). The subtype may do anything it likes to implement an operation, including offloading to hardware.

5. verify_signature().

   Optional. This is the entry point for signature verification. The subtype may do anything it likes to implement an operation, including offloading to hardware.

Instantiation Data Parsers

The asymmetric key type doesn't generally want to store or to deal with a raw blob of data that holds the key data. It would have to parse it and error check it each time it wanted to use it. Further, the contents of the blob may have various checks that can be performed on it (eg. self-signatures, validity dates) and may contain useful data about the key (identifiers, capabilities).

Also, the blob may represent a pointer to some hardware containing the key rather than the key itself.

Examples of blob formats for which parsers could be implemented include:

- OpenPGP packet stream [RFC 4880].
- X.509 ASN.1 stream.
- Pointer to TPM key.
- Pointer to UEFI key.
- PKCS#8 private key [RFC 5208].
- PKCS#5 encrypted private key [RFC 2898].

During key instantiation each parser in the list is tried until one doesn't return -EBADMSG.

The parser definition structure can be found in:

```
#include <keys/asymmetric-parser.h>
```

and looks like the following:

```
struct asymmetric_key_parser {
        struct module   *owner;
        const char      *name;

        int (*parse)(struct key_preparsed_payload *prep);
};
```

The owner and name fields should be set to the owning module and the name of the parser.

There is currently only a single operation defined by the parser, and it is mandatory:

1. parse().

   This is called to preparse the key from the key creation and update paths. In particular, it is called during the key creation _before_ a key is allocated, and as such, is permitted to provide the key's description in the case that the caller declines to do so.

   The caller passes a pointer to the following struct with all of the fields cleared, except for data, datalen and quotalen [see Documentation/security/keys/core.rst]:

   ```
   struct key_preparsed_payload {
           char            *description;
           void            *payload[4];
           const void      *data;
           size_t          datalen;
           size_t          quotalen;
   };
   ```

   The instantiation data is in a blob pointed to by data and is datalen in size. The parse() function is not permitted to change these two values at all, and shouldn't change any of the other values _unless_ they are recognise the blob format and will not return -EBADMSG to indicate it is not theirs.

If the parser is happy with the blob, it should propose a description for the key and attach it to ->description, ->payload[asym_subtype] should be set to point to the subtype to be used, ->payload[asym_crypto] should be set to point to the initialised data for that subtype, ->payload[asym_key_ids] should point to one or more hex fingerprints and quotalen should be updated to indicate how much quota this key should account for.

When clearing up, the data attached to ->payload[asym_key_ids] and ->description will be kfree()'d and the data attached to ->payload[asm_crypto] will be passed to the subtype's ->destroy() method to be disposed of. A module reference for the subtype pointed to by ->payload[asym_subtype] will be put.

If the data format is not recognised, -EBADMSG should be returned. If it is recognised, but the key cannot for some reason be set up, some other negative error code should be returned. On success, 0 should be returned.

The key's fingerprint string may be partially matched upon. For a public-key algorithm such as RSA and DSA this will likely be a printable hex version of the key's fingerprint.

Functions are provided to register and unregister parsers:

```
int register_asymmetric_key_parser(struct asymmetric_key_parser *parser);
void unregister_asymmetric_key_parser(struct asymmetric_key_parser *subtype);
```

Parsers may not have the same name. The names are otherwise only used for displaying in debugging messages.

Keyring Link Restrictions

Keyrings created from userspace using add_key can be configured to check the signature of the key being linked. Keys without a valid signature are not allowed to link.

Several restriction methods are available:

1. Restrict using the kernel builtin trusted keyring
    - Option string used with KEYCTL_RESTRICT_KEYRING: - "builtin_trusted"

   The kernel builtin trusted keyring will be searched for the signing key. If the builtin trusted keyring is not configured, all links will be rejected. The ca_keys kernel parameter also affects which keys are used for signature verification.
2. Restrict using the kernel builtin and secondary trusted keyrings
    - Option string used with KEYCTL_RESTRICT_KEYRING: - "builtin_and_secondary_trusted"

   The kernel builtin and secondary trusted keyrings will be searched for the signing key. If the secondary trusted keyring is not configured, this restriction will behave like the "builtin_trusted" option. The ca_keys kernel parameter also affects which keys are used for signature verification.
3. Restrict using a separate key or keyring
    - Option string used with KEYCTL_RESTRICT_KEYRING: - "key_or_keyring:<key or keyring serial number>[:chain]"

   Whenever a key link is requested, the link will only succeed if the key being linked is signed by one of the designated keys. This key may be specified directly by providing a serial number for one asymmetric key, or a group of keys may be searched for the signing key by providing the serial number for a keyring.

   When the "chain" option is provided at the end of the string, the keys within the destination keyring will also be searched for signing keys. This allows for verification of certificate chains by adding each certificate in order (starting closest to the root) to a keyring. For instance, one keyring can be populated with links to a set of root certificates, with a separate, restricted keyring set up for each certificate chain to be validated:

   ```
   # Create and populate a keyring for root certificates
   root_id=`keyctl add keyring root-certs "" @s`
   keyctl padd asymmetric "" $root_id < root1.cert
   keyctl padd asymmetric "" $root_id < root2.cert

   # Create and restrict a keyring for the certificate chain
   chain_id=`keyctl add keyring chain "" @s`
   keyctl restrict_keyring $chain_id asymmetric key_or_keyring:$root_id:chain

   # Attempt to add each certificate in the chain, starting with the
   # certificate closest to the root.
   keyctl padd asymmetric "" $chain_id < intermediateA.cert
   keyctl padd asymmetric "" $chain_id < intermediateB.cert
   keyctl padd asymmetric "" $chain_id < end-entity.cert
   ```

   If the final end-entity certificate is successfully added to the "chain" keyring, we can be certain that it has a valid signing chain going back to one of the root certificates.

   A single keyring can be used to verify a chain of signatures by restricting the keyring after linking the root

certificate:

```
# Create a keyring for the certificate chain and add the root
chain2_id=`keyctl add keyring chain2 "" @s`
keyctl padd asymmetric "" $chain2_id < root1.cert

# Restrict the keyring that already has root1.cert linked.  The cert
# will remain linked by the keyring.
keyctl restrict_keyring $chain2_id asymmetric key_or_keyring:0:chain

# Attempt to add each certificate in the chain, starting with the
# certificate closest to the root.
keyctl padd asymmetric "" $chain2_id < intermediateA.cert
keyctl padd asymmetric "" $chain2_id < intermediateB.cert
keyctl padd asymmetric "" $chain2_id < end-entity.cert
```

If the final end-entity certificate is successfully added to the "chain2" keyring, we can be certain that there is a valid signing chain going back to the root certificate that was added before the keyring was restricted.

In all of these cases, if the signing key is found the signature of the key to be linked will be verified using the signing key. The requested key is added to the keyring only if the signature is successfully verified. -ENOKEY is returned if the parent certificate could not be found, or -EKEYREJECTED is returned if the signature check fails or the key is blacklisted. Other errors may be returned if the signature check could not be performed.