Key Request Service

The key request service is part of the key retention service (refer to Documentation/security/keys/core.rst). This document explains more fully how the requesting algorithm works.

The process starts by either the kernel requesting a service by calling `request_key*()`:

```
struct key *request_key(const struct key_type *type,
                        const char *description,
                        const char *callout_info);
```

or:

```
struct key *request_key_tag(const struct key_type *type,
                            const char *description,
                            const struct key_tag *domain_tag,
                            const char *callout_info);
```

or:

```
struct key *request_key_with_auxdata(const struct key_type *type,
                                     const char *description,
                                     const struct key_tag *domain_tag,
                                     const char *callout_info,
                                     size_t callout_len,
                                     void *aux);
```

or:

```
struct key *request_key_rcu(const struct key_type *type,
                            const char *description,
                            const struct key_tag *domain_tag);
```

Or by userspace invoking the request_key system call:

```
key_serial_t request_key(const char *type,
                         const char *description,
                         const char *callout_info,
                         key_serial_t dest_keyring);
```

The main difference between the access points is that the in-kernel interface does not need to link the key to a keyring to prevent it from being immediately destroyed. The kernel interface returns a pointer directly to the key, and it's up to the caller to destroy the key.

The request_key_tag() call is like the in-kernel request_key(), except that it also takes a domain tag that allows keys to be separated by namespace and killed off as a group.

The request_key_with_auxdata() calls is like the request_key_tag() call, except that they permit auxiliary data to be passed to the upcaller (the default is NULL). This is only useful for those key types that define their own upcall mechanism rather than using /sbin/request-key.

The request_key_rcu() call is like the request_key_tag() call, except that it doesn't check for keys that are under construction and doesn't attempt to construct missing keys.

The userspace interface links the key to a keyring associated with the process to prevent the key from going away, and returns the serial number of the key to the caller.

The following example assumes that the key types involved don't define their own upcall mechanisms. If they do, then those should be substituted for the forking and execution of /sbin/request-key.

The Process

A request proceeds in the following manner:

1. Process A calls request_key() [the userspace syscall calls the kernel interface].
2. request_key() searches the process's subscribed keyrings to see if there's a suitable key there. If there is, it returns the key. If there isn't, and callout_info is not set, an error is returned. Otherwise the process proceeds to the next step.
3. request_key() sees that A doesn't have the desired key yet, so it creates two things:
    a. An uninstantiated key U of requested type and description.
    b. An authorisation key V that refers to key U and notes that process A is the context in which key U should be instantiated and secured, and from which associated key requests may be satisfied.

4. request_key() then forks and executes /sbin/request-key with a new session keyring that contains a link to auth key V.
5. /sbin/request-key assumes the authority associated with key U.
6. /sbin/request-key execs an appropriate program to perform the actual instantiation.
7. The program may want to access another key from A's context (say a Kerberos TGT key). It just requests the appropriate key, and the keyring search notes that the session keyring has auth key V in its bottom level.

   This will permit it to then search the keyrings of process A with the UID, GID, groups and security info of process A as if it was process A, and come up with key W.
8. The program then does what it must to get the data with which to instantiate key U, using key W as a reference (perhaps it contacts a Kerberos server using the TGT) and then instantiates key U.
9. Upon instantiating key U, auth key V is automatically revoked so that it may not be used again.
10. The program then exits 0 and request_key() deletes key V and returns key U to the caller.

This also extends further. If key W (step 7 above) didn't exist, key W would be created uninstantiated, another auth key (X) would be created (as per step 3) and another copy of /sbin/request-key spawned (as per step 4); but the context specified by auth key X will still be process A, as it was in auth key V.

This is because process A's keyrings can't simply be attached to /sbin/request-key at the appropriate places because (a) execve will discard two of them, and (b) it requires the same UID/GID/Groups all the way through.

Negative Instantiation And Rejection

Rather than instantiating a key, it is possible for the possessor of an authorisation key to negatively instantiate a key that's under construction. This is a short duration placeholder that causes any attempt at re-requesting the key while it exists to fail with error ENOKEY if negated or the specified error if rejected.

This is provided to prevent excessive repeated spawning of /sbin/request-key processes for a key that will never be obtainable.

Should the /sbin/request-key process exit anything other than 0 or die on a signal, the key under construction will be automatically negatively instantiated for a short amount of time.

The Search Algorithm

A search of any particular keyring proceeds in the following fashion:

1. When the key management code searches for a key (keyring_search_rcu) it firstly calls key_permission(SEARCH) on the keyring it's starting with, if this denies permission, it doesn't search further.
2. It considers all the non-keyring keys within that keyring and, if any key matches the criteria specified, calls key_permission(SEARCH) on it to see if the key is allowed to be found. If it is, that key is returned; if not, the search continues, and the error code is retained if of higher priority than the one currently set.
3. It then considers all the keyring-type keys in the keyring it's currently searching. It calls key_permission(SEARCH) on each keyring, and if this grants permission, it recurses, executing steps (2) and (3) on that keyring.

The process stops immediately a valid key is found with permission granted to use it. Any error from a previous match attempt is discarded and the key is returned.

When request_key() is invoked, if CONFIG_KEYS_REQUEST_CACHE=y, a per-task one-key cache is first checked for a match.

When search_process_keyrings() is invoked, it performs the following searches until one succeeds:

1. If extant, the process's thread keyring is searched.
2. If extant, the process's process keyring is searched.
3. The process's session keyring is searched.
4. If the process has assumed the authority associated with a request_key() authorisation key then:

   a. If extant, the calling process's thread keyring is searched.
   b. If extant, the calling process's process keyring is searched.
   c. The calling process's session keyring is searched.

The moment one succeeds, all pending errors are discarded and the found key is returned. If CONFIG_KEYS_REQUEST_CACHE=y, then that key is placed in the per-task cache, displacing the previous key. The cache is cleared on exit or just prior to resumption of userspace.

Only if all these fail does the whole thing fail with the highest priority error. Note that several errors may have come from LSM.

The error priority is:

```
EKEYREVOKED > EKEYEXPIRED > ENOKEY
```

EACCES/EPERM are only returned on a direct search of a specific keyring where the basal keyring does not grant Search permission.