Kernel Key Retention Service

This service allows cryptographic keys, authentication tokens, cross-domain user mappings, and similar to be cached in the kernel for the use of filesystems and other kernel services.

Keyrings are permitted; these are a special type of key that can hold links to other keys. Processes each have three standard keyring subscriptions that a kernel service can search for relevant keys.

The key service can be configured on by enabling:

"Security options"/"Enable access key retention support" (CONFIG_KEYS)

This document has the following sections:

Key Overview

In this context, keys represent units of cryptographic data, authentication tokens, keyrings, etc.. These are represented in the kernel by struct key.

Each key has a number of attributes:

- A serial number.
- A type.
- A description (for matching a key in a search).
- Access control information.
- An expiry time.
- A payload.
- State.

- Each key is issued a serial number of type key_serial_t that is unique for the lifetime of that key. All serial numbers are positive non-zero 32-bit integers.

  Userspace programs can use a key's serial numbers as a way to gain access to it, subject to permission checking.

- Each key is of a defined "type". Types must be registered inside the kernel by a kernel service (such as a filesystem) before keys of that type can be added or used. Userspace programs cannot define new types directly.

  Key types are represented in the kernel by struct key_type. This defines a number of operations that can be performed on a key of that type.

  Should a type be removed from the system, all the keys of that type will be invalidated.

- Each key has a description. This should be a printable string. The key type provides an operation to perform a match between the description on a key and a criterion string.

- Each key has an owner user ID, a group ID and a permissions mask. These are used to control what a process may do to a key from userspace, and whether a kernel service will be able to find the key.

- Each key can be set to expire at a specific time by the key type's instantiation function. Keys can also be immortal.

- Each key can have a payload. This is a quantity of data that represent the actual "key". In the case of a keyring, this is a list of keys to which the keyring links; in the case of a user-defined key, it's an arbitrary blob of data.

  Having a payload is not required; and the payload can, in fact, just be a value stored in the struct key itself.

  When a key is instantiated, the key type's instantiation function is called with a blob of data, and that then creates the key's payload in some way.

  Similarly, when userspace wants to read back the contents of the key, if permitted, another key type operation will be called to convert the key's attached payload back into a blob of data.

- Each key can be in one of a number of basic states:
    - Uninstantiated. The key exists, but does not have any data attached. Keys being requested from userspace will be in this state.
    - Instantiated. This is the normal state. The key is fully formed, and has data attached.
    - Negative. This is a relatively short-lived state. The key acts as a note saying that a previous call out to userspace failed, and acts as a throttle on key lookups. A negative key can be updated to a normal state.
    - Expired. Keys can have lifetimes set. If their lifetime is exceeded, they traverse to this state. An expired key can be updated back to a normal state.
    - Revoked. A key is put in this state by userspace action. It can't be found or operated upon (apart from by unlinking it).
    - Dead. The key's type was unregistered, and so the key is now useless.

Keys in the last three states are subject to garbage collection. See the section on "Garbage collection".

Key Service Overview

The key service provides a number of features besides keys:

- The key service defines three special key types:

  (+) "keyring"

  Keyrings are special keys that contain a list of other keys. Keyring lists can be modified using various system calls. Keyrings should not be given a payload when created.

  (+) "user"

  A key of this type has a description and a payload that are arbitrary blobs of data. These can be created, updated and read by userspace, and aren't intended for use by kernel services.

  (+) "logon"

  Like a "user" key, a "logon" key has a payload that is an arbitrary blob of data. It is intended as a place to store secrets which are accessible to the kernel but not to userspace programs.

  The description can be arbitrary, but must be prefixed with a non-zero length string that describes the key "subclass". The subclass is separated from the rest of the description by a ':'. "logon" keys can be created and updated from userspace, but the payload is only readable from kernel space.

- Each process subscribes to three keyrings: a thread-specific keyring, a process-specific keyring, and a session-specific keyring.

  The thread-specific keyring is discarded from the child when any sort of clone, fork, vfork or execve occurs. A new keyring is created only when required.

  The process-specific keyring is replaced with an empty one in the child on clone, fork, vfork unless CLONE_THREAD is supplied, in which case it is shared. execve also discards the process's process keyring and creates a new one.

  The session-specific keyring is persistent across clone, fork, vfork and execve, even when the latter executes a set-UID or set-GID binary. A process can, however, replace its current session keyring with a new one by using PR_JOIN_SESSION_KEYRING. It is permitted to request an anonymous new one, or to attempt to create or join one of a specific name.

  The ownership of the thread keyring changes when the real UID and GID of the thread changes.

- Each user ID resident in the system holds two special keyrings: a user specific keyring and a default user session keyring. The default session keyring is initialised with a link to the user-specific keyring.

  When a process changes its real UID, if it used to have no session key, it will be subscribed to the default session key for the new UID.

  If a process attempts to access its session key when it doesn't have one, it will be subscribed to the default for its current UID.

- Each user has two quotas against which the keys they own are tracked. One limits the total number of keys and keyrings, the other limits the total amount of description and payload space that can be consumed.

  The user can view information on this and other statistics through procfs files. The root user may also alter the quota limits through sysctl files (see the section "New procfs files").

  Process-specific and thread-specific keyrings are not counted towards a user's quota.

If a system call that modifies a key or keyring in some way would put the user over quota, the operation is refused and error EDQUOT is returned.

- There's a system call interface by which userspace programs can create and manipulate keys and keyrings.
- There's a kernel interface by which services can register types and search for keys.
- There's a way for the a search done from the kernel to call back to userspace to request a key that can't be found in a process's keyrings.
- An optional filesystem is available through which the key database can be viewed and manipulated.

Key Access Permissions

Keys have an owner user ID, a group access ID, and a permissions mask. The mask has up to eight bits each for possessor, user, group and other access. Only six of each set of eight bits are defined. These permissions granted are:

- View

  This permits a key or keyring's attributes to be viewed - including key type and description.

- Read

  This permits a key's payload to be viewed or a keyring's list of linked keys.

- Write

  This permits a key's payload to be instantiated or updated, or it allows a link to be added to or removed from a keyring.

- Search

  This permits keyrings to be searched and keys to be found. Searches can only recurse into nested keyrings that have search permission set.

- Link

  This permits a key or keyring to be linked to. To create a link from a keyring to a key, a process must have Write permission on the keyring and Link permission on the key.

- Set Attribute

  This permits a key's UID, GID and permissions mask to be changed.

For changing the ownership, group ID or permissions mask, being the owner of the key or having the sysadmin capability is sufficient.

SELinux Support

The security class "key" has been added to SELinux so that mandatory access controls can be applied to keys created within various contexts. This support is preliminary, and is likely to change quite significantly in the near future. Currently, all of the basic permissions explained above are provided in SELinux as well; SELinux is simply invoked after all basic permission checks have been performed.

The value of the file /proc/self/attr/keycreate influences the labeling of newly-created keys. If the contents of that file correspond to an SELinux security context, then the key will be assigned that context. Otherwise, the key will be assigned the current context of the task that invoked the key creation request. Tasks must be granted explicit permission to assign a particular context to newly-created keys, using the "create" permission in the key security class.

The default keyrings associated with users will be labeled with the default context of the user if and only if the login programs have been instrumented to properly initialize keycreate during the login process. Otherwise, they will be labeled with the context of the login program itself.

Note, however, that the default keyrings associated with the root user are labeled with the default kernel context, since they are created early in the boot process, before root has a chance to log in.

The keyrings associated with new threads are each labeled with the context of their associated thread, and both session and process keyrings are handled similarly.

New ProcFS Files

Two files have been added to procfs by which an administrator can find out about the status of the key service:

- /proc/keys

  This lists the keys that are currently viewable by the task reading the file, giving information about their type, description and permissions. It is not possible to view the payload of the key this way, though some information about it may be given.

  The only keys included in the list are those that grant View permission to the reading process whether or not it

possesses them. Note that LSM security checks are still performed, and may further filter out keys that the current process is not authorised to view.

The contents of the file look like this:

```
SERIAL   FLAGS  USAGE EXPY PERM     UID   GID   TYPE      DESCRIPTION: SUMMARY
00000001 I-----    39 perm 1f3f0000     0     0 keyring   _uid_ses.0: 1/4
00000002 I-----     2 perm 1f3f0000     0     0 keyring   _uid.0: empty
00000007 I-----     1 perm 1f3f0000     0     0 keyring   _pid.1: empty
0000018d I-----     1 perm 1f3f0000     0     0 keyring   _pid.412: empty
000004d2 I--Q--     1 perm 1f3f0000    32    -1 keyring   _uid.32: 1/4
000004d3 I--Q--     3 perm 1f3f0000    32    -1 keyring   _uid_ses.32: empty
00000892 I--QU-     1 perm 1f000000     0     0 user      metal:copper: 0
00000893 I--Q-N     1  35s 1f3f0000     0     0 user      metal:silver: 0
00000894 I--Q--     1  10h 003f0000     0     0 user      metal:gold: 0
```

The flags are:

```
I       Instantiated
R       Revoked
D       Dead
Q       Contributes to user's quota
U       Under construction by callback to userspace
N       Negative key
```

- /proc/key-users

  This file lists the tracking data for each user that has at least one key on the system. Such data includes quota information and statistics:

  ```
  [root@andromeda root]# cat /proc/key-users
  0:     46 45/45 1/100 13/10000
  29:     2 2/2 2/100 40/10000
  32:     2 2/2 2/100 40/10000
  38:     2 2/2 2/100 40/10000
  ```

  The format of each line is:

  ```
  <UID>:                  User ID to which this applies
  <usage>                 Structure refcount
  <inst>/<keys>           Total number of keys and number instantiated
  <keys>/<max>            Key count quota
  <bytes>/<max>           Key size quota
  ```

Four new sysctl files have been added also for the purpose of controlling the quota limits on keys:

- /proc/sys/kernel/keys/root_maxkeys /proc/sys/kernel/keys/root_maxbytes

  These files hold the maximum number of keys that root may have and the maximum total number of bytes of data that root may have stored in those keys.

- /proc/sys/kernel/keys/maxkeys /proc/sys/kernel/keys/maxbytes

  These files hold the maximum number of keys that each non-root user may have and the maximum total number of bytes of data that each of those users may have stored in their keys.

Root may alter these by writing each new limit as a decimal number string to the appropriate file.

Userspace System Call Interface

Userspace can manipulate keys directly through three new syscalls: add_key, request_key and keyctl. The latter provides a number of functions for manipulating keys.

When referring to a key directly, userspace programs should use the key's serial number (a positive 32-bit integer). However, there are some special values available for referring to special keys and keyrings that relate to the process making the call:

```
CONSTANT                        VALUE   KEY REFERENCED
==============================  ======  ===========================
KEY_SPEC_THREAD_KEYRING         -1      thread-specific keyring
KEY_SPEC_PROCESS_KEYRING        -2      process-specific keyring
KEY_SPEC_SESSION_KEYRING        -3      session-specific keyring
KEY_SPEC_USER_KEYRING           -4      UID-specific keyring
KEY_SPEC_USER_SESSION_KEYRING   -5      UID-session keyring
KEY_SPEC_GROUP_KEYRING          -6      GID-specific keyring
KEY_SPEC_REQKEY_AUTH_KEY        -7      assumed request_key()
                                          authorisation key
```

The main syscalls are:

- Create a new key of given type, description and payload and add it to the nominated keyring:

```
key_serial_t add_key(const char *type, const char *desc,
                     const void *payload, size_t plen,
                     key_serial_t keyring);
```

If a key of the same type and description as that proposed already exists in the keyring, this will try to update it with the given payload, or it will return error EEXIST if that function is not supported by the key type. The process must also have permission to write to the key to be able to update it. The new key will have all user permissions granted and no group or third party permissions.

Otherwise, this will attempt to create a new key of the specified type and description, and to instantiate it with the supplied payload and attach it to the keyring. In this case, an error will be generated if the process does not have permission to write to the keyring.

If the key type supports it, if the description is NULL or an empty string, the key type will try and generate a description from the content of the payload.

The payload is optional, and the pointer can be NULL if not required by the type. The payload is plen in size, and plen can be zero for an empty payload.

A new keyring can be generated by setting type "keyring", the keyring name as the description (or NULL) and setting the payload to NULL.

User defined keys can be created by specifying type "user". It is recommended that a user defined key's description by prefixed with a type ID and a colon, such as "krb5tgt:" for a Kerberos 5 ticket granting ticket.

Any other type must have been registered with the kernel in advance by a kernel service such as a filesystem.

The ID of the new or updated key is returned if successful.

- Search the process's keyrings for a key, potentially calling out to userspace to create it:

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

This function searches all the process's keyrings in the order thread, process, session for a matching key. This works very much like KEYCTL_SEARCH, including the optional attachment of the discovered key to a keyring.

If a key cannot be found, and if callout_info is not NULL, then /sbin/request-key will be invoked in an attempt to obtain a key. The callout_info string will be passed as an argument to the program.

To link a key into the destination keyring the key must grant link permission on the key to the caller and the keyring must grant write permission.

See also Documentation/security/keys/request-key.rst.

The keyctl syscall functions are:

- Map a special key ID to a real key ID for this process:

```
key_serial_t keyctl(KEYCTL_GET_KEYRING_ID, key_serial_t id,
                    int create);
```

The special key specified by "id" is looked up (with the key being created if necessary) and the ID of the key or keyring thus found is returned if it exists.

If the key does not yet exist, the key will be created if "create" is non-zero; and the error ENOKEY will be returned if "create" is zero.

- Replace the session keyring this process subscribes to with a new one:

```
key_serial_t keyctl(KEYCTL_JOIN_SESSION_KEYRING, const char *name);
```

If name is NULL, an anonymous keyring is created attached to the process as its session keyring, displacing the old session keyring.

If name is not NULL, if a keyring of that name exists, the process attempts to attach it as the session keyring, returning an error if that is not permitted; otherwise a new keyring of that name is created and attached as the session keyring.

To attach to a named keyring, the keyring must have search permission for the process's ownership.

The ID of the new session keyring is returned if successful.

- Update the specified key:

```
long keyctl(KEYCTL_UPDATE, key_serial_t key, const void *payload,
            size_t plen);
```

This will try to update the specified key with the given payload, or it will return error EOPNOTSUPP if that function is not supported by the key type. The process must also have permission to write to the key to be able to update it.

The payload is of length plen, and may be absent or empty as for add_key().

- Revoke a key:

```
long keyctl(KEYCTL_REVOKE, key_serial_t key);
```

This makes a key unavailable for further operations. Further attempts to use the key will be met with error EKEYREVOKED, and the key will no longer be findable.

- Change the ownership of a key:

```
long keyctl(KEYCTL_CHOWN, key_serial_t key, uid_t uid, gid_t gid);
```

This function permits a key's owner and group ID to be changed. Either one of uid or gid can be set to -1 to suppress that change.

Only the superuser can change a key's owner to something other than the key's current owner. Similarly, only the superuser can change a key's group ID to something other than the calling process's group ID or one of its group list members.

- Change the permissions mask on a key:

```
long keyctl(KEYCTL_SETPERM, key_serial_t key, key_perm_t perm);
```

This function permits the owner of a key or the superuser to change the permissions mask on a key.

Only bits the available bits are permitted; if any other bits are set, error EINVAL will be returned.

- Describe a key:

```
long keyctl(KEYCTL_DESCRIBE, key_serial_t key, char *buffer,
            size_t buflen);
```

This function returns a summary of the key's attributes (but not its payload data) as a string in the buffer provided.

Unless there's an error, it always returns the amount of data it could produce, even if that's too big for the buffer, but it won't copy more than requested to userspace. If the buffer pointer is NULL then no copy will take place.

A process must have view permission on the key for this function to be successful.

If successful, a string is placed in the buffer in the following format:

```
<type>;<uid>;<gid>;<perm>;<description>
```

Where type and description are strings, uid and gid are decimal, and perm is hexadecimal. A NUL character is included at the end of the string if the buffer is sufficiently big.

This can be parsed with:

```
sscanf(buffer, "%[^;];%d;%d;%o;%s", type, &uid, &gid, &mode, desc);
```

- Clear out a keyring:

```
long keyctl(KEYCTL_CLEAR, key_serial_t keyring);
```

This function clears the list of keys attached to a keyring. The calling process must have write permission on the keyring, and it must be a keyring (or else error ENOTDIR will result).

This function can also be used to clear special kernel keyrings if they are appropriately marked if the user has CAP_SYS_ADMIN capability. The DNS resolver cache keyring is an example of this.

- Link a key into a keyring:

```
long keyctl(KEYCTL_LINK, key_serial_t keyring, key_serial_t key);
```

This function creates a link from the keyring to the key. The process must have write permission on the keyring and must have link permission on the key.

Should the keyring not be a keyring, error ENOTDIR will result; and if the keyring is full, error ENFILE will result.

The link procedure checks the nesting of the keyrings, returning ELOOP if it appears too deep or EDEADLK if the link would introduce a cycle.

Any links within the keyring to keys that match the new key in terms of type and description will be discarded from the keyring as the new one is added.

- Move a key from one keyring to another:

```
long keyctl(KEYCTL_MOVE,
            key_serial_t id,
            key_serial_t from_ring_id,
            key_serial_t to_ring_id,
            unsigned int flags);
```

Move the key specified by "id" from the keyring specified by "from_ring_id" to the keyring specified by "to_ring_id". If the two keyrings are the same, nothing is done.

"flags" can have KEYCTL_MOVE_EXCL set in it to cause the operation to fail with EEXIST if a matching key exists in the destination keyring, otherwise such a key will be replaced.

A process must have link permission on the key for this function to be successful and write permission on both keyrings. Any errors that can occur from KEYCTL_LINK also apply on the destination keyring here.

- Unlink a key or keyring from another keyring:

```
long keyctl(KEYCTL_UNLINK, key_serial_t keyring, key_serial_t key);
```

This function looks through the keyring for the first link to the specified key, and removes it if found. Subsequent links to that key are ignored. The process must have write permission on the keyring.

If the keyring is not a keyring, error ENOTDIR will result; and if the key is not present, error ENOENT will be the result.

- Search a keyring tree for a key:

```
key_serial_t keyctl(KEYCTL_SEARCH, key_serial_t keyring,
                    const char *type, const char *description,
                    key_serial_t dest_keyring);
```

This searches the keyring tree headed by the specified keyring until a key is found that matches the type and description criteria. Each keyring is checked for keys before recursion into its children occurs.

The process must have search permission on the top level keyring, or else error EACCES will result. Only keyrings that the process has search permission on will be recursed into, and only keys and keyrings for which a process has search permission can be matched. If the specified keyring is not a keyring, ENOTDIR will result.

If the search succeeds, the function will attempt to link the found key into the destination keyring if one is supplied (non-zero ID). All the constraints applicable to KEYCTL_LINK apply in this case too.

Error ENOKEY, EKEYREVOKED or EKEYEXPIRED will be returned if the search fails. On success, the resulting key ID will be returned.

- Read the payload data from a key:

```
long keyctl(KEYCTL_READ, key_serial_t keyring, char *buffer,
            size_t buflen);
```

This function attempts to read the payload data from the specified key into the buffer. The process must have read permission on the key to succeed.

The returned data will be processed for presentation by the key type. For instance, a keyring will return an array of key_serial_t entries representing the IDs of all the keys to which it is subscribed. The user defined key type will return its data as is. If a key type does not implement this function, error EOPNOTSUPP will result.

If the specified buffer is too small, then the size of the buffer required will be returned. Note that in this case, the contents of the buffer may have been overwritten in some undefined way.

Otherwise, on success, the function will return the amount of data copied into the buffer.

- Instantiate a partially constructed key:

```
long keyctl(KEYCTL_INSTANTIATE, key_serial_t key,
            const void *payload, size_t plen,
            key_serial_t keyring);
long keyctl(KEYCTL_INSTANTIATE_IOV, key_serial_t key,
            const struct iovec *payload_iov, unsigned ioc,
            key_serial_t keyring);
```

If the kernel calls back to userspace to complete the instantiation of a key, userspace should use this call to supply data for the key before the invoked process returns, or else the key will be marked negative automatically.

The process must have write access on the key to be able to instantiate it, and the key must be uninstantiated.

If a keyring is specified (non-zero), the key will also be linked into that keyring, however all the constraints applying in KEYCTL_LINK apply in this case too.

The payload and plen arguments describe the payload data as for add_key().

The payload_iov and ioc arguments describe the payload data in an iovec array instead of a single buffer.

- Negatively instantiate a partially constructed key:

```
long keyctl(KEYCTL_NEGATE, key_serial_t key,
            unsigned timeout, key_serial_t keyring);
long keyctl(KEYCTL_REJECT, key_serial_t key,
            unsigned timeout, unsigned error, key_serial_t keyring);
```

If the kernel calls back to userspace to complete the instantiation of a key, userspace should use this call mark the key as negative before the invoked process returns if it is unable to fulfill the request.

The process must have write access on the key to be able to instantiate it, and the key must be uninstantiated.

If a keyring is specified (non-zero), the key will also be linked into that keyring, however all the constraints applying in KEYCTL_LINK apply in this case too.

If the key is rejected, future searches for it will return the specified error code until the rejected key expires. Negating the key is the same as rejecting the key with ENOKEY as the error code.

- Set the default request-key destination keyring:

```
long keyctl(KEYCTL_SET_REQKEY_KEYRING, int reqkey_defl);
```

This sets the default keyring to which implicitly requested keys will be attached for this thread. reqkey_defl should be one of these constants:

CONSTANT	VALUE	NEW DEFAULT KEYRING
KEY_REQKEY_DEFL_NO_CHANGE	-1	No change
KEY_REQKEY_DEFL_DEFAULT	0	Default[1]
KEY_REQKEY_DEFL_THREAD_KEYRING	1	Thread keyring
KEY_REQKEY_DEFL_PROCESS_KEYRING	2	Process keyring
KEY_REQKEY_DEFL_SESSION_KEYRING	3	Session keyring
KEY_REQKEY_DEFL_USER_KEYRING	4	User keyring
KEY_REQKEY_DEFL_USER_SESSION_KEYRING	5	User session keyring
KEY_REQKEY_DEFL_GROUP_KEYRING	6	Group keyring

The old default will be returned if successful and error EINVAL will be returned if reqkey_defl is not one of the above values.

The default keyring can be overridden by the keyring indicated to the request_key() system call.

Note that this setting is inherited across fork/exec.

[1] The default is: the thread keyring if there is one, otherwise the process keyring if there is one, otherwise the session keyring if there is one, otherwise the user default session keyring.

- Set the timeout on a key:

```
long keyctl(KEYCTL_SET_TIMEOUT, key_serial_t key, unsigned timeout);
```

This sets or clears the timeout on a key. The timeout can be 0 to clear the timeout or a number of seconds to set the expiry time that far into the future.

The process must have attribute modification access on a key to set its timeout. Timeouts may not be set with this function on negative, revoked or expired keys.

- Assume the authority granted to instantiate a key:

```
long keyctl(KEYCTL_ASSUME_AUTHORITY, key_serial_t key);
```

This assumes or divests the authority required to instantiate the specified key. Authority can only be assumed if the thread has the authorisation key associated with the specified key in its keyrings somewhere.

Once authority is assumed, searches for keys will also search the requester's keyrings using the requester's security label, UID, GID and groups.

If the requested authority is unavailable, error EPERM will be returned, likewise if the authority has been revoked because the target key is already instantiated.

If the specified key is 0, then any assumed authority will be divested.

The assumed authoritative key is inherited across fork and exec.

- Get the LSM security context attached to a key:

```
long keyctl(KEYCTL_GET_SECURITY, key_serial_t key, char *buffer,
            size_t buflen)
```

This function returns a string that represents the LSM security context attached to a key in the buffer provided.

Unless there's an error, it always returns the amount of data it could produce, even if that's too big for the buffer, but it won't copy more than requested to userspace. If the buffer pointer is NULL then no copy will take place.

A NUL character is included at the end of the string if the buffer is sufficiently big. This is included in the returned count. If no LSM is in force then an empty string will be returned.

A process must have view permission on the key for this function to be successful.

- Install the calling process's session keyring on its parent:

```
long keyctl(KEYCTL_SESSION_TO_PARENT);
```

This functions attempts to install the calling process's session keyring on to the calling process's parent, replacing the parent's current session keyring.

The calling process must have the same ownership as its parent, the keyring must have the same ownership as the calling process, the calling process must have LINK permission on the keyring and the active LSM module mustn't deny permission, otherwise error EPERM will be returned.

Error ENOMEM will be returned if there was insufficient memory to complete the operation, otherwise 0 will be returned to indicate success.

The keyring will be replaced next time the parent process leaves the kernel and resumes executing userspace.

- Invalidate a key:

```
long keyctl(KEYCTL_INVALIDATE, key_serial_t key);
```

This function marks a key as being invalidated and then wakes up the garbage collector. The garbage collector immediately removes invalidated keys from all keyrings and deletes the key when its reference count reaches zero.

Keys that are marked invalidated become invisible to normal key operations immediately, though they are still visible in /proc/keys until deleted (they're marked with an 'i' flag).

A process must have search permission on the key for this function to be successful.

- Compute a Diffie-Hellman shared secret or public key:

```
long keyctl(KEYCTL_DH_COMPUTE, struct keyctl_dh_params *params,
            char *buffer, size_t buflen, struct keyctl_kdf_params *kdf);
```

The params struct contains serial numbers for three keys:

```
- The prime, p, known to both parties
- The local private key
- The base integer, which is either a shared generator or the
  remote public key
```

The value computed is:

```
result = base ^ private (mod prime)
```

If the base is the shared generator, the result is the local public key. If the base is the remote public key, the result is the shared secret.

If the parameter kdf is NULL, the following applies:

  - The buffer length must be at least the length of the prime, or zero.
  - If the buffer length is nonzero, the length of the result is returned when it is successfully calculated and copied in to the buffer. When the buffer length is zero, the minimum required buffer length is returned.

The kdf parameter allows the caller to apply a key derivation function (KDF) on the Diffie-Hellman computation where only the result of the KDF is returned to the caller. The KDF is characterized with struct keyctl_kdf_params as follows:

  - `char *hashname` specifies the NUL terminated string identifying the hash used from the kernel crypto API and applied for the KDF operation. The KDF implemenation complies with SP800-56A as well as with SP800-108 (the counter KDF).
  - `char *otherinfo` specifies the OtherInfo data as documented in SP800-56A section 5.8.1.2. The length of the buffer is given with otherinfolen. The format of OtherInfo is defined by the caller. The otherinfo pointer may be NULL if no OtherInfo shall be used.

This function will return error EOPNOTSUPP if the key type is not supported, error ENOKEY if the key could not be found, or error EACCES if the key is not readable by the caller. In addition, the function will return EMSGSIZE when the parameter kdf is non-NULL and either the buffer length or the OtherInfo length exceeds the allowed length.

- Restrict keyring linkage:

```
long keyctl(KEYCTL_RESTRICT_KEYRING, key_serial_t keyring,
            const char *type, const char *restriction);
```

An existing keyring can restrict linkage of additional keys by evaluating the contents of the key according to a restriction scheme.

"keyring" is the key ID for an existing keyring to apply a restriction to. It may be empty or may already have keys linked. Existing linked keys will remain in the keyring even if the new restriction would reject them.

"type" is a registered key type.

"restriction" is a string describing how key linkage is to be restricted. The format varies depending on the key type, and the string is passed to the lookup_restriction() function for the requested type. It may specify a method and relevant data for the restriction such as signature verification or constraints on key payload. If the requested key type is later unregistered, no keys may be added to the keyring after the key type is removed.

To apply a keyring restriction the process must have Set Attribute permission and the keyring must not be previously restricted.

One application of restricted keyrings is to verify X.509 certificate chains or individual certificate signatures using the asymmetric key type. See Documentation/crypto/asymmetric-keys.rst for specific restrictions applicable to the asymmetric key type.

- Query an asymmetric key:

```
long keyctl(KEYCTL_PKEY_QUERY,
            key_serial_t key_id, unsigned long reserved,
            const char *params,
            struct keyctl_pkey_query *info);
```

Get information about an asymmetric key. Specific algorithms and encodings may be queried by using the `params` argument. This is a string containing a space- or tab-separated string of key-value pairs. Currently supported keys include `enc` and `hash`. The information is returned in the keyctl_pkey_query struct:

```
__u32   supported_ops;
__u32   key_size;
__u16   max_data_size;
__u16   max_sig_size;
__u16   max_enc_size;
__u16   max_dec_size;
__u32   __spare[10];
```

`supported_ops` contains a bit mask of flags indicating which ops are supported. This is constructed from a bitwise-OR of:

```
KEYCTL_SUPPORTS_{ENCRYPT,DECRYPT,SIGN,VERIFY}
```

`key_size` indicated the size of the key in bits.

`max_*_size` indicate the maximum sizes in bytes of a blob of data to be signed, a signature blob, a blob to be encrypted and a blob to be decrypted.

`__spare[]` must be set to 0. This is intended for future use to hand over one or more passphrases needed unlock a key.

If successful, 0 is returned. If the key is not an asymmetric key, EOPNOTSUPP is returned.

- Encrypt, decrypt, sign or verify a blob using an asymmetric key:

```
long keyctl(KEYCTL_PKEY_ENCRYPT,
            const struct keyctl_pkey_params *params,
            const char *info,
            const void *in,
            void *out);

long keyctl(KEYCTL_PKEY_DECRYPT,
            const struct keyctl_pkey_params *params,
            const char *info,
            const void *in,
            void *out);

long keyctl(KEYCTL_PKEY_SIGN,
            const struct keyctl_pkey_params *params,
            const char *info,
            const void *in,
            void *out);

long keyctl(KEYCTL_PKEY_VERIFY,
            const struct keyctl_pkey_params *params,
            const char *info,
            const void *in,
            const void *in2);
```

Use an asymmetric key to perform a public-key cryptographic operation a blob of data. For encryption and verification, the asymmetric key may only need the public parts to be available, but for decryption and signing the private parts are required also.

The parameter block pointed to by params contains a number of integer values:

```
__s32           key_id;
```

```
__u32           in_len;
__u32           out_len;
__u32           in2_len;
```

`key_id` is the ID of the asymmetric key to be used. `in_len` and `in2_len` indicate the amount of data in the in and in2 buffers and `out_len` indicates the size of the out buffer as appropriate for the above operations.

For a given operation, the in and out buffers are used as follows:

```
Operation ID            in,in_len       out,out_len     in2,in2_len
======================= =============== =============== ===============
KEYCTL_PKEY_ENCRYPT     Raw data        Encrypted data  -
KEYCTL_PKEY_DECRYPT     Encrypted data  Raw data        -
KEYCTL_PKEY_SIGN        Raw data        Signature       -
KEYCTL_PKEY_VERIFY      Raw data        -               Signature
```

`info` is a string of key=value pairs that supply supplementary information. These include:

`enc=<encoding>` The encoding of the encrypted/signature blob. This can be "pkcs1" for RSASSA-PKCS1-v1.5 or RSAES-PKCS1-v1.5; "pss" for "RSASSA-PSS"; "oaep" for "RSAES-OAEP". If omitted or is "raw", the raw output of the encryption function is specified.

`hash=<algo>` If the data buffer contains the output of a hash function and the encoding includes some indication of which hash function was used, the hash function can be specified with this, eg. "hash=sha256".

The `__spare[]` space in the parameter block must be set to 0. This is intended, amongst other things, to allow the passing of passphrases required to unlock a key.

If successful, encrypt, decrypt and sign all return the amount of data written into the output buffer. Verification returns 0 on success.

- Watch a key or keyring for changes:

```
long keyctl(KEYCTL_WATCH_KEY, key_serial_t key, int queue_fd,
            const struct watch_notification_filter *filter);
```

This will set or remove a watch for changes on the specified key or keyring.

"key" is the ID of the key to be watched.

"queue_fd" is a file descriptor referring to an open pipe which manages the buffer into which notifications will be delivered.

"filter" is either NULL to remove a watch or a filter specification to indicate what events are required from the key.

See Documentation/watch_queue.rst for more information.

Note that only one watch may be emplaced for any particular { key, queue_fd } combination.

Notification records look like:

```
struct key_notification {
        struct watch_notification watch;
        __u32   key_id;
        __u32   aux;
};
```

In this, watch::type will be "WATCH_TYPE_KEY_NOTIFY" and subtype will be one of:

```
NOTIFY_KEY_INSTANTIATED
NOTIFY_KEY_UPDATED
NOTIFY_KEY_LINKED
NOTIFY_KEY_UNLINKED
NOTIFY_KEY_CLEARED
NOTIFY_KEY_REVOKED
NOTIFY_KEY_INVALIDATED
NOTIFY_KEY_SETATTR
```

Where these indicate a key being instantiated/rejected, updated, a link being made in a keyring, a link being removed from a keyring, a keyring being cleared, a key being revoked, a key being invalidated or a key having one of its attributes changed (user, group, perm, timeout, restriction).

If a watched key is deleted, a basic watch_notification will be issued with "type" set to WATCH_TYPE_META and "subtype" set to watch_meta_removal_notification. The watchpoint ID will be set in the "info" field.

This needs to be configured by enabling:

"Provide key/keyring change notifications" (KEY_NOTIFICATIONS)

Kernel Services

The kernel services for key management are fairly simple to deal with. They can be broken down into two areas: keys and key types.

Dealing with keys is fairly straightforward. Firstly, the kernel service registers its type, then it searches for a key of that type. It should retain the key as long as it has need of it, and then it should release it. For a filesystem or device file, a search would probably be performed during the open call, and the key released upon close. How to deal with conflicting keys due to two different users opening the same file is left to the filesystem author to solve.

To access the key manager, the following header must be #included:

```
<linux/key.h>
```

Specific key types should have a header file under include/keys/ that should be used to access that type. For keys of type "user", for example, that would be:

```
<keys/user-type.h>
```

Note that there are two different types of pointers to keys that may be encountered:

- struct key *

  This simply points to the key structure itself. Key structures will be at least four-byte aligned.

- key_ref_t

  This is equivalent to a `struct key *`, but the least significant bit is set if the caller "possesses" the key. By "possession" it is meant that the calling processes has a searchable link to the key from one of its keyrings. There are three functions for dealing with these:

  ```
  key_ref_t make_key_ref(const struct key *key, bool possession);

  struct key *key_ref_to_ptr(const key_ref_t key_ref);

  bool is_key_possessed(const key_ref_t key_ref);
  ```

  The first function constructs a key reference from a key pointer and possession information (which must be true or false).

  The second function retrieves the key pointer from a reference and the third retrieves the possession flag.

When accessing a key's payload contents, certain precautions must be taken to prevent access vs modification races. See the section "Notes on accessing payload contents" for more information.

- To search for a key, call:

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

  This is used to request a key or keyring with a description that matches the description specified according to the key type's match_preparse() method. This permits approximate matching to occur. If callout_string is not NULL, then /sbin/request-key will be invoked in an attempt to obtain the key from userspace. In that case, callout_string will be passed as an argument to the program.

  Should the function fail error ENOKEY, EKEYEXPIRED or EKEYREVOKED will be returned.

  If successful, the key will have been attached to the default keyring for implicitly obtained request-key keys, as set by KEYCTL_SET_REQKEY_KEYRING.

  See also Documentation/security/keys/request-key.rst.

- To search for a key in a specific domain, call:

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

  This is identical to request_key(), except that a domain tag may be specifies that causes search algorithm to only match keys matching that tag. The domain_tag may be NULL, specifying a global domain that is separate from any nominated domain.

- To search for a key, passing auxiliary data to the upcaller, call:

  ```
  struct key *request_key_with_auxdata(const struct key_type *type,
                                       const char *description,
                                       struct key_tag *domain_tag,
                                       const void *callout_info,
  ```

```
                                        size_t callout_len,
                                        void *aux);
```

This is identical to request_key_tag(), except that the auxiliary data is passed to the key_type->request_key() op if it exists, and the callout_info is a blob of length callout_len, if given (the length may be 0).

- To search for a key under RCU conditions, call:

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

which is similar to request_key_tag() except that it does not check for keys that are under construction and it will not call out to userspace to construct a key if it can't find a match.

- When it is no longer required, the key should be released using:

```
void key_put(struct key *key);
```

Or:

```
void key_ref_put(key_ref_t key_ref);
```

These can be called from interrupt context. If CONFIG_KEYS is not set then the argument will not be parsed.

- Extra references can be made to a key by calling one of the following functions:

```
struct key *__key_get(struct key *key);
struct key *key_get(struct key *key);
```

Keys so references will need to be disposed of by calling key_put() when they've been finished with. The key pointer passed in will be returned.

In the case of key_get(), if the pointer is NULL or CONFIG_KEYS is not set then the key will not be dereferenced and no increment will take place.

- A key's serial number can be obtained by calling:

```
key_serial_t key_serial(struct key *key);
```

If key is NULL or if CONFIG_KEYS is not set then 0 will be returned (in the latter case without parsing the argument).

- If a keyring was found in the search, this can be further searched by:

```
key_ref_t keyring_search(key_ref_t keyring_ref,
                         const struct key_type *type,
                         const char *description,
                         bool recurse)
```

This searches the specified keyring only (recurse == false) or keyring tree (recurse == true) specified for a matching key. Error ENOKEY is returned upon failure (use IS_ERR/PTR_ERR to determine). If successful, the returned key will need to be released.

The possession attribute from the keyring reference is used to control access through the permissions mask and is propagated to the returned key reference pointer if successful.

- A keyring can be created by:

```
struct key *keyring_alloc(const char *description, uid_t uid, gid_t gid,
                          const struct cred *cred,
                          key_perm_t perm,
                          struct key_restriction *restrict_link,
                          unsigned long flags,
                          struct key *dest);
```

This creates a keyring with the given attributes and returns it. If dest is not NULL, the new keyring will be linked into the keyring to which it points. No permission checks are made upon the destination keyring.

Error EDQUOT can be returned if the keyring would overload the quota (pass KEY_ALLOC_NOT_IN_QUOTA in flags if the keyring shouldn't be accounted towards the user's quota). Error ENOMEM can also be returned.

If restrict_link is not NULL, it should point to a structure that contains the function that will be called each time an attempt is made to link a key into the new keyring. The structure may also contain a key pointer and an associated key type. The function is called to check whether a key may be added into the keyring or not. The key type is used by the garbage collector to clean up function or data pointers in this structure if the given key type is unregistered. Callers of key_create_or_update() within the kernel can pass KEY_ALLOC_BYPASS_RESTRICTION to suppress the check. An example of using this is to manage rings of cryptographic keys that are set up when the kernel boots where userspace is also permitted to add keys - provided they can be verified by a key the kernel

already has.

When called, the restriction function will be passed the keyring being added to, the key type, the payload of the key being added, and data to be used in the restriction check. Note that when a new key is being created, this is called between payload preparsing and actual key creation. The function should return 0 to allow the link or an error to reject it.

A convenience function, restrict_link_reject, exists to always return -EPERM to in this case.

- To check the validity of a key, this function can be called:

  ```
  int validate_key(struct key *key);
  ```

  This checks that the key in question hasn't expired or and hasn't been revoked. Should the key be invalid, error EKEYEXPIRED or EKEYREVOKED will be returned. If the key is NULL or if CONFIG_KEYS is not set then 0 will be returned (in the latter case without parsing the argument).

- To register a key type, the following function should be called:

  ```
  int register_key_type(struct key_type *type);
  ```

  This will return error EEXIST if a type of the same name is already present.

- To unregister a key type, call:

  ```
  void unregister_key_type(struct key_type *type);
  ```

Under some circumstances, it may be desirable to deal with a bundle of keys. The facility provides access to the keyring type for managing such a bundle:

```
struct key_type key_type_keyring;
```

This can be used with a function such as request_key() to find a specific keyring in a process's keyrings. A keyring thus found can then be searched with keyring_search(). Note that it is not possible to use request_key() to search a specific keyring, so using keyrings in this way is of limited utility.

Notes On Accessing Payload Contents

The simplest payload is just data stored in key->payload directly. In this case, there's no need to indulge in RCU or locking when accessing the payload.

More complex payload contents must be allocated and pointers to them set in the key->payload.data[] array. One of the following ways must be selected to access the data:

1. Unmodifiable key type.

   If the key type does not have a modify method, then the key's payload can be accessed without any form of locking, provided that it's known to be instantiated (uninstantiated keys cannot be "found").

2. The key's semaphore.

   The semaphore could be used to govern access to the payload and to control the payload pointer. It must be write-locked for modifications and would have to be read-locked for general access. The disadvantage of doing this is that the accessor may be required to sleep.

3. RCU.

   RCU must be used when the semaphore isn't already held; if the semaphore is held then the contents can't change under you unexpectedly as the semaphore must still be used to serialise modifications to the key. The key management code takes care of this for the key type.

   However, this means using:

   ```
   rcu_read_lock() ... rcu_dereference() ... rcu_read_unlock()
   ```

   to read the pointer, and:

   ```
   rcu_dereference() ... rcu_assign_pointer() ... call_rcu()
   ```

   to set the pointer and dispose of the old contents after a grace period. Note that only the key type should ever modify a key's payload.

   Furthermore, an RCU controlled payload must hold a struct rcu_head for the use of call_rcu() and, if the payload is of variable size, the length of the payload. key->datalen cannot be relied upon to be consistent with the payload just dereferenced if the key's semaphore is not held.

   Note that key->payload.data[0] has a shadow that is marked for __rcu usage. This is called key->payload.rcu_data0. The following accessors wrap the RCU calls to this element:

a. Set or change the first payload pointer:

```
rcu_assign_keypointer(struct key *key, void *data);
```

b. Read the first payload pointer with the key semaphore held:

```
[const] void *dereference_key_locked([const] struct key *key);

Note that the return value will inherit its constness from the key
parameter.  Static analysis will give an error if it things the lock
isn't held.
```

c. Read the first payload pointer with the RCU read lock held:

```
const void *dereference_key_rcu(const struct key *key);
```

Defining a Key Type

A kernel service may want to define its own key type. For instance, an AFS filesystem might want to define a Kerberos 5 ticket key type. To do this, it author fills in a key_type struct and registers it with the system.

Source files that implement key types should include the following header file:

```
<linux/key-type.h>
```

The structure has a number of fields, some of which are mandatory:

- `const char *name`

  The name of the key type. This is used to translate a key type name supplied by userspace into a pointer to the structure.

- `size_t def_datalen`

  This is optional - it supplies the default payload data length as contributed to the quota. If the key type's payload is always or almost always the same size, then this is a more efficient way to do things.

  The data length (and quota) on a particular key can always be changed during instantiation or update by calling:

  ```
  int key_payload_reserve(struct key *key, size_t datalen);
  ```

  With the revised data length. Error EDQUOT will be returned if this is not viable.

- `int (*vet_description)(const char *description);`

  This optional method is called to vet a key description. If the key type doesn't approve of the key description, it may return an error, otherwise it should return 0.

- `int (*preparse)(struct key_preparsed_payload *prep);`

  This optional method permits the key type to attempt to parse payload before a key is created (add key) or the key semaphore is taken (update or instantiate key). The structure pointed to by prep looks like:

  ```
  struct key_preparsed_payload {
          char            *description;
          union key_payload payload;
          const void      *data;
          size_t          datalen;
          size_t          quotalen;
          time_t          expiry;
  };
  ```

  Before calling the method, the caller will fill in data and datalen with the payload blob parameters; quotalen will be filled in with the default quota size from the key type; expiry will be set to TIME_T_MAX and the rest will be cleared.

  If a description can be proposed from the payload contents, that should be attached as a string to the description field. This will be used for the key description if the caller of add_key() passes NULL or "".

  The method can attach anything it likes to payload. This is merely passed along to the instantiate() or update() operations. If set, the expiry time will be applied to the key if it is instantiated from this data.

  The method should return 0 if successful or a negative error code otherwise.

- `void (*free_preparse)(struct key_preparsed_payload *prep);`

  This method is only required if the preparse() method is provided, otherwise it is unused. It cleans up anything attached to the description and payload fields of the key_preparsed_payload struct as filled in by the preparse() method. It will always be called after preparse() returns successfully, even if instantiate() or update() succeed.

- `int (*instantiate)(struct key *key, struct key_preparsed_payload *prep);`

This method is called to attach a payload to a key during construction. The payload attached need not bear any relation to the data passed to this function.

The prep->data and prep->datalen fields will define the original payload blob. If preparse() was supplied then other fields may be filled in also.

If the amount of data attached to the key differs from the size in keytype->def_datalen, then key_payload_reserve() should be called.

This method does not have to lock the key in order to attach a payload. The fact that KEY_FLAG_INSTANTIATED is not set in key->flags prevents anything else from gaining access to the key.

It is safe to sleep in this method.

generic_key_instantiate() is provided to simply copy the data from prep->payload.data[] to key->payload.data[], with RCU-safe assignment on the first element. It will then clear prep->payload.data[] so that the free_preparse method doesn't release the data.

- `int (*update)(struct key *key, const void *data, size_t datalen);`

  If this type of key can be updated, then this method should be provided. It is called to update a key's payload from the blob of data provided.

  The prep->data and prep->datalen fields will define the original payload blob. If preparse() was supplied then other fields may be filled in also.

  key_payload_reserve() should be called if the data length might change before any changes are actually made. Note that if this succeeds, the type is committed to changing the key because it's already been altered, so all memory allocation must be done first.

  The key will have its semaphore write-locked before this method is called, but this only deters other writers; any changes to the key's payload must be made under RCU conditions, and call_rcu() must be used to dispose of the old payload.

  key_payload_reserve() should be called before the changes are made, but after all allocations and other potentially failing function calls are made.

  It is safe to sleep in this method.

- `int (*match_preparse)(struct key_match_data *match_data);`

  This method is optional. It is called when a key search is about to be performed. It is given the following structure:

  ```
  struct key_match_data {
          bool (*cmp)(const struct key *key,
                      const struct key_match_data *match_data);
          const void      *raw_data;
          void            *preparsed;
          unsigned        lookup_type;
  };
  ```

  On entry, raw_data will be pointing to the criteria to be used in matching a key by the caller and should not be modified. `(*cmp)()` will be pointing to the default matcher function (which does an exact description match against raw_data) and lookup_type will be set to indicate a direct lookup.

  The following lookup_type values are available:

  - KEYRING_SEARCH_LOOKUP_DIRECT - A direct lookup hashes the type and description to narrow down the search to a small number of keys.
  - KEYRING_SEARCH_LOOKUP_ITERATE - An iterative lookup walks all the keys in the keyring until one is matched. This must be used for any search that's not doing a simple direct match on the key description.

  The method may set cmp to point to a function of its choice that does some other form of match, may set lookup_type to KEYRING_SEARCH_LOOKUP_ITERATE and may attach something to the preparsed pointer for use by `(*cmp)()`. `(*cmp)()` should return true if a key matches and false otherwise.

  If preparsed is set, it may be necessary to use the match_free() method to clean it up.

  The method should return 0 if successful or a negative error code otherwise.

  It is permitted to sleep in this method, but `(*cmp)()` may not sleep as locks will be held over it.

  If match_preparse() is not provided, keys of this type will be matched exactly by their description.

- `void (*match_free)(struct key_match_data *match_data);`

  This method is optional. If given, it called to clean up match_data->preparsed after a successful call to match_preparse().

- `void (*revoke)(struct key *key);`

This method is optional. It is called to discard part of the payload data upon a key being revoked. The caller will have the key semaphore write-locked.

It is safe to sleep in this method, though care should be taken to avoid a deadlock against the key semaphore.

- `void (*destroy)(struct key *key);`

  This method is optional. It is called to discard the payload data on a key when it is being destroyed.

  This method does not need to lock the key to access the payload; it can consider the key as being inaccessible at this time. Note that the key's type may have been changed before this function is called.

  It is not safe to sleep in this method; the caller may hold spinlocks.

- `void (*describe)(const struct key *key, struct seq_file *p);`

  This method is optional. It is called during /proc/keys reading to summarise a key's description and payload in text form.

  This method will be called with the RCU read lock held. rcu_dereference() should be used to read the payload pointer if the payload is to be accessed. key->datalen cannot be trusted to stay consistent with the contents of the payload.

  The description will not change, though the key's state may.

  It is not safe to sleep in this method; the RCU read lock is held by the caller.

- `long (*read)(const struct key *key, char __user *buffer, size_t buflen);`

  This method is optional. It is called by KEYCTL_READ to translate the key's payload into something a blob of data for userspace to deal with. Ideally, the blob should be in the same format as that passed in to the instantiate and update methods.

  If successful, the blob size that could be produced should be returned rather than the size copied.

  This method will be called with the key's semaphore read-locked. This will prevent the key's payload changing. It is not necessary to use RCU locking when accessing the key's payload. It is safe to sleep in this method, such as might happen when the userspace buffer is accessed.

- `int (*request_key)(struct key_construction *cons, const char *op, void *aux);`

  This method is optional. If provided, request_key() and friends will invoke this function rather than upcalling to /sbin/request-key to operate upon a key of this type.

  The aux parameter is as passed to request_key_async_with_auxdata() and similar or is NULL otherwise. Also passed are the construction record for the key to be operated upon and the operation type (currently only "create").

  This method is permitted to return before the upcall is complete, but the following function must be called under all circumstances to complete the instantiation process, whether or not it succeeds, whether or not there's an error:

  ```
  void complete_request_key(struct key_construction *cons, int error);
  ```

  The error parameter should be 0 on success, -ve on error. The construction record is destroyed by this action and the authorisation key will be revoked. If an error is indicated, the key under construction will be negatively instantiated if it wasn't already instantiated.

  If this method returns an error, that error will be returned to the caller of request_key*(). complete_request_key() must be called prior to returning.

  The key under construction and the authorisation key can be found in the key_construction struct pointed to by cons:

  - `struct key *key;`

    The key under construction.

  - `struct key *authkey;`

    The authorisation key.

- `struct key_restriction *(*lookup_restriction)(const char *params);`

  This optional method is used to enable userspace configuration of keyring restrictions. The restriction parameter string (not including the key type name) is passed in, and this method returns a pointer to a key_restriction structure containing the relevant functions and data to evaluate each attempted key link operation. If there is no match, -EINVAL is returned.

- `asym_eds_op` and `asym_verify_signature`:

  ```
  int (*asym_eds_op)(struct kernel_pkey_params *params,
                     const void *in, void *out);
  int (*asym_verify_signature)(struct kernel_pkey_params *params,
  ```

```
                                const void *in, const void *in2);
```

These methods are optional. If provided the first allows a key to be used to encrypt, decrypt or sign a blob of data, and the second allows a key to verify a signature.

In all cases, the following information is provided in the params block:

```
struct kernel_pkey_params {
        struct key      *key;
        const char      *encoding;
        const char      *hash_algo;
        char            *info;
        __u32           in_len;
        union {
                __u32   out_len;
                __u32   in2_len;
        };
        enum kernel_pkey_operation op : 8;
};
```

This includes the key to be used; a string indicating the encoding to use (for instance, "pkcs1" may be used with an RSA key to indicate RSASSA-PKCS1-v1.5 or RSAES-PKCS1-v1.5 encoding or "raw" if no encoding); the name of the hash algorithm used to generate the data for a signature (if appropriate); the sizes of the input and output (or second input) buffers; and the ID of the operation to be performed.

For a given operation ID, the input and output buffers are used as follows:

```
Operation ID            in,in_len       out,out_len     in2,in2_len
======================= =============== =============== ===============
kernel_pkey_encrypt     Raw data        Encrypted data  -
kernel_pkey_decrypt     Encrypted data  Raw data        -
kernel_pkey_sign        Raw data        Signature       -
kernel_pkey_verify      Raw data        -               Signature
```

asym_eds_op() deals with encryption, decryption and signature creation as specified by params->op. Note that params->op is also set for asym_verify_signature().

Encrypting and signature creation both take raw data in the input buffer and return the encrypted result in the output buffer. Padding may have been added if an encoding was set. In the case of signature creation, depending on the encoding, the padding created may need to indicate the digest algorithm - the name of which should be supplied in hash_algo.

Decryption takes encrypted data in the input buffer and returns the raw data in the output buffer. Padding will get checked and stripped off if an encoding was set.

Verification takes raw data in the input buffer and the signature in the second input buffer and checks that the one matches the other. Padding will be validated. Depending on the encoding, the digest algorithm used to generate the raw data may need to be indicated in hash_algo.

If successful, asym_eds_op() should return the number of bytes written into the output buffer. asym_verify_signature() should return 0.

A variety of errors may be returned, including EOPNOTSUPP if the operation is not supported; EKEYREJECTED if verification fails; ENOPKG if the required crypto isn't available.

- `asym_query`:

```
int (*asym_query)(const struct kernel_pkey_params *params,
                  struct kernel_pkey_query *info);
```

This method is optional. If provided it allows information about the public or asymmetric key held in the key to be determined.

The parameter block is as for asym_eds_op() and co. but in_len and out_len are unused. The encoding and hash_algo fields should be used to reduce the returned buffer/data sizes as appropriate.

If successful, the following information is filled in:

```
struct kernel_pkey_query {
        __u32           supported_ops;
        __u32           key_size;
        __u16           max_data_size;
        __u16           max_sig_size;
        __u16           max_enc_size;
        __u16           max_dec_size;
};
```

The supported_ops field will contain a bitmask indicating what operations are supported by the key, including encryption of a blob, decryption of a blob, signing a blob and verifying the signature on a blob. The following constants are defined for this:

KEYCTL_SUPPORTS_{ENCRYPT,DECRYPT,SIGN,VERIFY}

The key_size field is the size of the key in bits. max_data_size and max_sig_size are the maximum raw data and signature sizes for creation and verification of a signature; max_enc_size and max_dec_size are the maximum raw data and signature sizes for encryption and decryption. The max_*_size fields are measured in bytes.

If successful, 0 will be returned. If the key doesn't support this, EOPNOTSUPP will be returned.

Request-Key Callback Service

To create a new key, the kernel will attempt to execute the following command line:

```
/sbin/request-key create <key> <uid> <gid> \
        <threadring> <processring> <sessionring> <callout_info>
```

<key> is the key being constructed, and the three keyrings are the process keyrings from the process that caused the search to be issued. These are included for two reasons:

1 There may be an authentication token in one of the keyrings that is required to obtain the key, eg: a Kerberos Ticket-Granting Ticket.

2 The new key should probably be cached in one of these rings.

This program should set it UID and GID to those specified before attempting to access any more keys. It may then look around for a user specific process to hand the request off to (perhaps a path held in placed in another key by, for example, the KDE desktop manager).

The program (or whatever it calls) should finish construction of the key by calling KEYCTL_INSTANTIATE or KEYCTL_INSTANTIATE_IOV, which also permits it to cache the key in one of the keyrings (probably the session ring) before returning. Alternatively, the key can be marked as negative with KEYCTL_NEGATE or KEYCTL_REJECT; this also permits the key to be cached in one of the keyrings.

If it returns with the key remaining in the unconstructed state, the key will be marked as being negative, it will be added to the session keyring, and an error will be returned to the key requestor.

Supplementary information may be provided from whoever or whatever invoked this service. This will be passed as the <callout_info> parameter. If no such information was made available, then "-" will be passed as this parameter instead.

Similarly, the kernel may attempt to update an expired or a soon to expire key by executing:

```
/sbin/request-key update <key> <uid> <gid> \
        <threadring> <processring> <sessionring>
```

In this case, the program isn't required to actually attach the key to a ring; the rings are provided for reference.

Garbage Collection

Dead keys (for which the type has been removed) will be automatically unlinked from those keyrings that point to them and deleted as soon as possible by a background garbage collector.

Similarly, revoked and expired keys will be garbage collected, but only after a certain amount of time has passed. This time is set as a number of seconds in:

```
/proc/sys/kernel/keys/gc_delay
```