NAME

keyctl − manipulate the kernel's key management facility

SYNOPSIS

#include <sys/types.h>
#include <keyutils.h>

long keyctl(int *operation*, **...)**

/* For direct call via syscall(2): */
#include <asm/unistd.h>
#include <linux/keyctl.h>
#include <unistd.h>

long syscall(__NR_keyctl, int *operation*, **__kernel_ulong_t** *arg2*,
    __kernel_ulong_t *arg3*, **__kernel_ulong_t** *arg4*,
    __kernel_ulong_t *arg5*);

No glibc wrapper is provided for this system call; see NOTES.

DESCRIPTION

keyctl() allows user-space programs to perform key manipulation.

The operation performed by **keyctl**() is determined by the value of the *operation* argument. Each of these operations is wrapped by the *libkeyutils* library (provided by the *keyutils* package) into individual functions (noted below) to permit the compiler to check types.

The permitted values for *operation* are:

KEYCTL_GET_KEYRING_ID (since Linux 2.6.10)

Map a special key ID to a real key ID for this process.

This operation looks up the special key whose ID is provided in *arg2* (cast to *key_serial_t*). If the special key is found, the ID of the corresponding real key is returned as the function result. The following values may be specified in *arg2*:

KEY_SPEC_THREAD_KEYRING

This specifies the calling thread's thread-specific keyring. See **thread-keyring**(7).

KEY_SPEC_PROCESS_KEYRING

This specifies the caller's process-specific keyring. See **process-keyring**(7).

KEY_SPEC_SESSION_KEYRING

This specifies the caller's session-specific keyring. See **session-keyring**(7).

KEY_SPEC_USER_KEYRING

This specifies the caller's UID-specific keyring. See **user-keyring**(7).

KEY_SPEC_USER_SESSION_KEYRING

This specifies the caller's UID-session keyring. See **user-session-keyring**(7).

KEY_SPEC_REQKEY_AUTH_KEY (since Linux 2.6.16)

This specifies the authorization key created by **request_key**(2) and passed to the process it spawns to generate a key. This key is available only in a **request-key**(8)-style program that was passed an authorization key by the kernel and ceases to be available once the requested key has been instantiated; see **request_key**(2).

KEY_SPEC_REQUESTOR_KEYRING (since Linux 2.6.29)

This specifies the key ID for the **request_key**(2) destination keyring. This keyring is available only in a **request-key**(8)-style program that was passed an authorization key by the kernel and ceases to be available once the requested key has been instantiated; see **request_key**(2).

The behavior if the key specified in *arg2* does not exist depends on the value of *arg3* (cast to *int*). If *arg3* contains a nonzero value, then—if it is appropriate to do so (e.g., when looking up the user,

user-session, or session key)—a new key is created and its real key ID returned as the function result. Otherwise, the operation fails with the error **ENOKEY**.

If a valid key ID is specified in *arg2*, and the key exists, then this operation simply returns the key ID. If the key does not exist, the call fails with error **ENOKEY**.

The caller must have *search* permission on a keyring in order for it to be found.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_get_keyring_ID**(3).

KEYCTL_JOIN_SESSION_KEYRING (since Linux 2.6.10)

Replace the session keyring this process subscribes to with a new session keyring.

If *arg2* is NULL, an anonymous keyring with the description "_ses" is created and the process is subscribed to that keyring as its session keyring, displacing the previous session keyring.

Otherwise, *arg2* (cast to *char **) is treated as the description (name) of a keyring, and the behavior is as follows:

* If a keyring with a matching description exists, the process will attempt to subscribe to that keyring as its session keyring if possible; if that is not possible, an error is returned. In order to subscribe to the keyring, the caller must have *search* permission on the keyring.

* If a keyring with a matching description does not exist, then a new keyring with the specified description is created, and the process is subscribed to that keyring as its session keyring.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_join_session_keyring**(3).

KEYCTL_UPDATE (since Linux 2.6.10)

Update a key's data payload.

The *arg2* argument (cast to *key_serial_t*) specifies the ID of the key to be updated. The *arg3* argument (cast to *void **) points to the new payload and *arg4* (cast to *size_t*) contains the new payload size in bytes.

The caller must have *write* permission on the key specified and the key type must support updating.

A negatively instantiated key (see the description of **KEYCTL_REJECT**) can be positively instantiated with this operation.

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_update**(3).

KEYCTL_REVOKE (since Linux 2.6.10)

Revoke the key with the ID provided in *arg2* (cast to *key_serial_t*). The key is scheduled for garbage collection; it will no longer be findable, and will be unavailable for further operations. Further attempts to use the key will fail with the error **EKEYREVOKED**.

The caller must have *write* or *setattr* permission on the key.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_revoke**(3).

KEYCTL_CHOWN (since Linux 2.6.10)

Change the ownership (user and group ID) of a key.

The *arg2* argument (cast to *key_serial_t*) contains the key ID. The *arg3* argument (cast to *uid_t*) contains the new user ID (or −1 in case the user ID shouldn't be changed). The *arg4* argument (cast to *gid_t*) contains the new group ID (or −1 in case the group ID shouldn't be changed).

The key must grant the caller *setattr* permission.

For the UID to be changed, or for the GID to be changed to a group the caller is not a member of, the caller must have the **CAP_SYS_ADMIN** capability (see **capabilities**(7)).

If the UID is to be changed, the new user must have sufficient quota to accept the key. The quota deduction will be removed from the old user to the new user should the UID be changed.

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_chown**(3).

KEYCTL_SETPERM (since Linux 2.6.10)

Change the permissions of the key with the ID provided in the *arg2* argument (cast to *key_serial_t*) to the permissions provided in the *arg3* argument (cast to *key_perm_t*).

If the caller doesn't have the **CAP_SYS_ADMIN** capability, it can change permissions only for the keys it owns. (More precisely: the caller's filesystem UID must match the UID of the key.)

The key must grant *setattr* permission to the caller *regardless* of the caller's capabilities.

The permissions in *arg3* specify masks of available operations for each of the following user categories:

possessor (since Linux 2.6.14)
This is the permission granted to a process that possesses the key (has it attached searchably to one of the process's keyrings); see **keyrings**(7).

user This is the permission granted to a process whose filesystem UID matches the UID of the key.

group This is the permission granted to a process whose filesystem GID or any of its supplementary GIDs matches the GID of the key.

other This is the permission granted to other processes that do not match the *user* and *group* categories.

The *user*, *group*, and *other* categories are exclusive: if a process matches the *user* category, it will not receive permissions granted in the *group* category; if a process matches the *user* or *group* category, then it will not receive permissions granted in the *other* category.

The *possessor* category grants permissions that are cumulative with the grants from the *user*, *group*, or *other* category.

Each permission mask is eight bits in size, with only six bits currently used. The available permissions are:

view This permission allows reading attributes of a key.

This permission is required for the **KEYCTL_DESCRIBE** operation.

The permission bits for each category are **KEY_POS_VIEW**, **KEY_USR_VIEW**, **KEY_GRP_VIEW**, and **KEY_OTH_VIEW**.

read This permission allows reading a key's payload.

This permission is required for the **KEYCTL_READ** operation.

The permission bits for each category are **KEY_POS_READ**, **KEY_USR_READ**, **KEY_GRP_READ**, and **KEY_OTH_READ**.

write This permission allows update or instantiation of a key's payload. For a keyring, it allows keys to be linked and unlinked from the keyring,

This permission is required for the **KEYCTL_UPDATE**, **KEYCTL_REVOKE**, **KEYCTL_CLEAR**, **KEYCTL_LINK**, and **KEYCTL_UNLINK** operations.

The permission bits for each category are **KEY_POS_WRITE**, **KEY_USR_WRITE**, **KEY_GRP_WRITE**, and **KEY_OTH_WRITE**.

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

This permission is required for the **KEYCTL_GET_KEYRING_ID**, **KEYCTL_JOIN_SESSION_KEYRING**, **KEYCTL_SEARCH**, and **KEYCTL_INVALIDATE** operations.

The permission bits for each category are **KEY_POS_SEARCH**, **KEY_USR_SEARCH**, **KEY_GRP_SEARCH**, and **KEY_OTH_SEARCH**.

link This permission allows a key or keyring to be linked to.

This permission is required for the **KEYCTL_LINK** and **KEYCTL_SESSION_TO_PARENT** operations.

The permission bits for each category are **KEY_POS_LINK**, **KEY_USR_LINK**, **KEY_GRP_LINK**, and **KEY_OTH_LINK**.

setattr (since Linux 2.6.15).
This permission allows a key's UID, GID, and permissions mask to be changed.

This permission is required for the **KEYCTL_REVOKE**, **KEYCTL_CHOWN**, and **KEYCTL_SETPERM** operations.

The permission bits for each category are **KEY_POS_SETATTR**, **KEY_USR_SETATTR**, **KEY_GRP_SETATTR**, and **KEY_OTH_SETATTR**.

As a convenience, the following macros are defined as masks for all of the permission bits in each of the user categories: **KEY_POS_ALL**, **KEY_USR_ALL**, **KEY_GRP_ALL**, and **KEY_OTH_ALL**.

The *arg4* and *arg5* arguments are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_setperm**(3).

KEYCTL_DESCRIBE (since Linux 2.6.10)
Obtain a string describing the attributes of a specified key.

The ID of the key to be described is specified in *arg2* (cast to *key_serial_t*). The descriptive string is returned in the buffer pointed to by *arg3* (cast to *char* *); *arg4* (cast to *size_t*) specifies the size of that buffer in bytes.

The key must grant the caller *view* permission.

The returned string is null-terminated and contains the following information about the key:

type;*uid*;*gid*;*perm*;*description*

In the above, *type* and *description* are strings, *uid* and *gid* are decimal strings, and *perm* is a hexadecimal permissions mask. The descriptive string is written with the following format:

```
%s;%d;%d;%08x;%s
```

Note: the intention is that the descriptive string should be extensible in future kernel versions. In particular, the *description* field will not contain semicolons; it should be parsed by working backwards from the end of the string to find the last semicolon. This allows future semicolon-delimited fields to be inserted in the descriptive string in the future.

Writing to the buffer is attempted only when *arg3* is non-NULL and the specified buffer size is large enough to accept the descriptive string (including the terminating null byte). In order to determine whether the buffer size was too small, check to see if the return value of the operation is greater than *arg4*.

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_describe**(3).

KEYCTL_CLEAR

Clear the contents of (i.e., unlink all keys from) a keyring.

The ID of the key (which must be of keyring type) is provided in *arg2* (cast to *key_serial_t*).

The caller must have *write* permission on the keyring.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_clear**(3).

KEYCTL_LINK (since Linux 2.6.10)

Create a link from a keyring to a key.

The key to be linked is specified in *arg2* (cast to *key_serial_t*); the keyring is specified in *arg3* (cast to *key_serial_t*).

If a key with the same type and description is already linked in the keyring, then that key is displaced from the keyring.

Before creating the link, the kernel checks the nesting of the keyrings and returns appropriate errors if the link would produce a cycle or if the nesting of keyrings would be too deep (The limit on the nesting of keyrings is determined by the kernel constant **KEYRING_SEARCH_MAX_DEPTH**, defined with the value 6, and is necessary to prevent overflows on the kernel stack when recursively searching keyrings).

The caller must have *link* permission on the key being added and *write* permission on the keyring.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_link**(3).

KEYCTL_UNLINK (since Linux 2.6.10)

Unlink a key from a keyring.

The ID of the key to be unlinked is specified in *arg2* (cast to *key_serial_t*); the ID of the keyring from which it is to be unlinked is specified in *arg3* (cast to *key_serial_t*).

If the key is not currently linked into the keyring, an error results.

The caller must have *write* permission on the keyring from which the key is being removed.

If the last link to a key is removed, then that key will be scheduled for destruction.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_unlink**(3).

KEYCTL_SEARCH (since Linux 2.6.10)

Search for a key in a keyring tree, returning its ID and optionally linking it to a specified keyring.

The tree to be searched is specified by passing the ID of the head keyring in *arg2* (cast to *key_serial_t*). The search is performed breadth-first and recursively.

The *arg3* and *arg4* arguments specify the key to be searched for: *arg3* (cast as *char* *) contains the key type (a null-terminated character string up to 32 bytes in size, including the terminating null byte), and *arg4* (cast as *char* *) contains the description of the key (a null-terminated character string up to 4096 bytes in size, including the terminating null byte).

The source keyring must grant *search* permission to the caller. When performing the recursive search, only keyrings that grant the caller *search* permission will be searched. Only keys with for which the caller has *search* permission can be found.

If the key is found, its ID is returned as the function result.

If the key is found and *arg5* (cast to *key_serial_t*) is nonzero, then, subject to the same constraints and rules as **KEYCTL_LINK**, the key is linked into the keyring whose ID is specified in *arg5*. If the destination keyring specified in *arg5* already contains a link to a key that has the same type and description, then that link will be displaced by a link to the key found by this operation.

Instead of valid existing keyring IDs, the source (*arg2*) and destination (*arg5*) keyrings can be one of the special keyring IDs listed under **KEYCTL_GET_KEYRING_ID**.

This operation is exposed by *libkeyutils* via the function **keyctl_search**(3).

KEYCTL_READ (since Linux 2.6.10)

Read the payload data of a key.

The ID of the key whose payload is to be read is specified in *arg2* (cast to *key_serial_t*). This can be the ID of an existing key, or any of the special key IDs listed for **KEYCTL_GET_KEYRING_ID**.

The payload is placed in the buffer pointed by *arg3* (cast to *char* *); the size of that buffer must be specified in *arg4* (cast to *size_t*).

The returned data will be processed for presentation according to the key type. For example, a keyring will return an array of *key_serial_t* entries representing the IDs of all the keys that are linked to it. The *user* key type will return its data as is. If a key type does not implement this function, the operation fails with the error **EOPNOTSUPP**.

If *arg3* is not NULL, as much of the payload data as will fit is copied into the buffer. On a successful return, the return value is always the total size of the payload data. To determine whether the buffer was of sufficient size, check to see that the return value is less than or equal to the value supplied in *arg4*.

The key must either grant the caller *read* permission, or grant the caller *search* permission when searched for from the process keyrings (i.e., the key is possessed).

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_read**(3).

KEYCTL_INSTANTIATE (since Linux 2.6.10)

(Positively) instantiate an uninstantiated key with a specified payload.

The ID of the key to be instantiated is provided in *arg2* (cast to *key_serial_t*).

The key payload is specified in the buffer pointed to by *arg3* (cast to *void* *); the size of that buffer is specified in *arg4* (cast to *size_t*).

The payload may be a NULL pointer and the buffer size may be 0 if this is supported by the key type (e.g., it is a keyring).

The operation may be fail if the payload data is in the wrong format or is otherwise invalid.

If *arg5* (cast to *key_serial_t*) is nonzero, then, subject to the same constraints and rules as **KEYCTL_LINK**, the instantiated key is linked into the keyring whose ID specified in *arg5*.

The caller must have the appropriate authorization key, and once the uninstantiated key has been instantiated, the authorization key is revoked. In other words, this operation is available only from a **request-key**(8)-style program. See **request_key**(2) for an explanation of uninstantiated keys and key instantiation.

This operation is exposed by *libkeyutils* via the function **keyctl_instantiate**(3).

KEYCTL_NEGATE (since Linux 2.6.10)

Negatively instantiate an uninstantiated key.

This operation is equivalent to the call:

```
keyctl(KEYCTL_REJECT, arg2, arg3, ENOKEY, arg4);
```

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_negate**(3).

KEYCTL_SET_REQKEY_KEYRING (since Linux 2.6.13)

Set the default keyring to which implicitly requested keys will be linked for this thread, and return the previous setting. Implicit key requests are those made by internal kernel components, such as can occur when, for example, opening files on an AFS or NFS filesystem. Setting the default keyring also has an effect when requesting a key from user space; see **request_key**(2) for details.

The *arg2* argument (cast to *int*) should contain one of the following values, to specify the new default keyring:

KEY_REQKEY_DEFL_NO_CHANGE

Don't change the default keyring. This can be used to discover the current default keyring (without changing it).

KEY_REQKEY_DEFL_DEFAULT

This selects the default behaviour, which is to use the thread-specific keyring if there is one, otherwise the process-specific keyring if there is one, otherwise the session keyring if there is one, otherwise the UID-specific session keyring, otherwise the user-specific keyring.

KEY_REQKEY_DEFL_THREAD_KEYRING

Use the thread-specific keyring (**thread-keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_PROCESS_KEYRING

Use the process-specific keyring (**process-keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_SESSION_KEYRING

Use the session-specific keyring (**session-keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_USER_KEYRING

Use the UID-specific keyring (**user-keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_USER_SESSION_KEYRING

Use the UID-specific session keyring (**user-session-keyring**(7)) as the new default keyring.

KEY_REQKEY_DEFL_REQUESTOR_KEYRING (since Linux 2.6.29)

Use the requestor keyring.

All other values are invalid.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

The setting controlled by this operation is inherited by the child of **fork**(2) and preserved across **execve**(2).

This operation is exposed by *libkeyutils* via the function **keyctl_set_reqkey_keyring**(3).

KEYCTL_SET_TIMEOUT (since Linux 2.6.16)

Set a timeout on a key.

The ID of the key is specified in *arg2* (cast to *key_serial_t*). The timeout value, in seconds from the current time, is specified in *arg3* (cast to *unsigned int*). The timeout is measured against the realtime clock.

Specifying the timeout value as 0 clears any existing timeout on the key.

The */proc/keys* file displays the remaining time until each key will expire. (This is the only method of discovering the timeout on a key.)

The caller must either have the *setattr* permission on the key or hold an instantiation authorization token for the key (see **request_key**(2)).

The key and any links to the key will be automatically garbage collected after the timeout expires. Subsequent attempts to access the key will then fail with the error **EKEYEXPIRED**.

This operation cannot be used to set timeouts on revoked, expired, or negatively instantiated keys.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_set_timeout**(3).

KEYCTL_ASSUME_AUTHORITY (since Linux 2.6.16)

Assume (or divest) the authority for the calling thread to instantiate a key.

The *arg2* argument (cast to *key_serial_t*) specifies either a nonzero key ID to assume authority, or the value 0 to divest authority.

If *arg2* is nonzero, then it specifies the ID of an uninstantiated key for which authority is to be assumed. That key can then be instantiated using one of **KEYCTL_INSTANTIATE**, **KEYCTL_INSTANTIATE_IOV**, **KEYCTL_REJECT**, or **KEYCTL_NEGATE**. Once the key has been instantiated, the thread is automatically divested of authority to instantiate the key.

Authority over a key can be assumed only if the calling thread has present in its keyrings the authorization key that is associated with the specified key. (In other words, the **KEYCTL_ASSUME_AUTHORITY** operation is available only from a **request-key**(8)-style program; see **request_key**(2) for an explanation of how this operation is used.) The caller must have *search* permission on the authorization key.

If the specified key has a matching authorization key, then the ID of that key is returned. The authorization key can be read (**KEYCTL_READ**) to obtain the callout information passed to **request_key**(2).

If the ID given in *arg2* is 0, then the currently assumed authority is cleared (divested), and the value 0 is returned.

The **KEYCTL_ASSUME_AUTHORITY** mechanism allows a program such as **request-key**(8) to assume the necessary authority to instantiate a new uninstantiated key that was created as a consequence of a call to **request_key**(2). For further information, see **request_key**(2) and the kernel source file *Documentation/security/keys-request-key.txt*.

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_assume_authority**(3).

KEYCTL_GET_SECURITY (since Linux 2.6.26)

Get the LSM (Linux Security Module) security label of the specified key.

The ID of the key whose security label is to be fetched is specified in *arg2* (cast to *key_serial_t*). The security label (terminated by a null byte) will be placed in the buffer pointed to by *arg3* argument (cast to *char **); the size of the buffer must be provided in *arg4* (cast to *size_t*).

If *arg3* is specified as NULL or the buffer size specified in *arg4* is too small, the full size of the security label string (including the terminating null byte) is returned as the function result, and nothing is copied to the buffer.

The caller must have *view* permission on the specified key.

The returned security label string will be rendered in a form appropriate to the LSM in force. For example, with SELinux, it may look like:

```
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

If no LSM is currently in force, then an empty string is placed in the buffer.

The *arg5* argument is ignored.

This operation is exposed by *libkeyutils* via the functions **keyctl_get_security**(3) and **keyctl_get_security_alloc**(3).

KEYCTL_SESSION_TO_PARENT (since Linux 2.6.32)

Replace the session keyring to which the *parent* of the calling process subscribes with the session keyring of the calling process.

The keyring will be replaced in the parent process at the point where the parent next transitions from kernel space to user space.

The keyring must exist and must grant the caller *link* permission. The parent process must be single-threaded and have the same effective ownership as this process and must not be set-user-ID or set-group-ID. The UID of the parent process's existing session keyring (f it has one), as well as the UID of the caller's session keyring much match the caller's effective UID.

The fact that it is the parent process that is affected by this operation allows a program such as the shell to start a child process that uses this operation to change the shell's session keyring. (This is what the **keyctl**(1) **new_session** command does.)

The arguments *arg2*, *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_session_to_parent**(3).

KEYCTL_REJECT (since Linux 2.6.39)

Mark a key as negatively instantiated and set an expiration timer on the key. This operation provides a superset of the functionality of the earlier **KEYCTL_NEGATE** operation.

The ID of the key that is to be negatively instantiated is specified in *arg2* (cast to *key_serial_t*). The *arg3* (cast to *unsigned int*) argument specifies the lifetime of the key, in seconds. The *arg4* argument (cast to *unsigned int*) specifies the error to be returned when a search hits this key; typically, this is one of **EKEYREJECTED**, **EKEYREVOKED**, or **EKEYEXPIRED**.

If *arg5* (cast to *key_serial_t*) is nonzero, then, subject to the same constraints and rules as **KEYCTL_LINK**, the negatively instantiated key is linked into the keyring whose ID is specified in *arg5*.

The caller must have the appropriate authorization key. In other words, this operation is available only from a **request-key**(8)-style program. See **request_key**(2).

The caller must have the appropriate authorization key, and once the uninstantiated key has been instantiated, the authorization key is revoked. In other words, this operation is available only from a **request-key**(8)-style program. See **request_key**(2) for an explanation of uninstantiated keys and key instantiation.

This operation is exposed by *libkeyutils* via the function **keyctl_reject**(3).

KEYCTL_INSTANTIATE_IOV (since Linux 2.6.39)

Instantiate an uninstantiated key with a payload specified via a vector of buffers.

This operation is the same as **KEYCTL_INSTANTIATE**, but the payload data is specified as an array of *iovec* structures:

```
struct iovec {
    void  *iov_base;    /* Starting address of buffer */
    size_t iov_len;     /* Size of buffer (in bytes) */
};
```

The pointer to the payload vector is specified in *arg3* (cast as *const struct iovec* *). The number of items in the vector is specified in *arg4* (cast as *unsigned int*).

The *arg2* (key ID) and *arg5* (keyring ID) are interpreted as for **KEYCTL_INSTANTIATE**.

This operation is exposed by *libkeyutils* via the function **keyctl_instantiate_iov**(3).

KEYCTL_INVALIDATE (since Linux 3.5)

Mark a key as invalid.

The ID of the key to be invalidated is specified in *arg2* (cast to *key_serial_t*).

To invalidate a key, the caller must have *search* permission on the key.

This operation marks the key as invalid and schedules immediate garbage collection. The garbage collector removes the invalidated key from all keyrings and deletes the key when its reference count reaches zero. After this operation, the key will be ignored by all searches, even if it is not yet deleted.

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

The arguments *arg3*, *arg4*, and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_invalidate**(3).

KEYCTL_GET_PERSISTENT (since Linux 3.13)

Get the persistent keyring (**persistent-keyring**(7)) for a specified user and link it to a specified keyring.

The user ID is specified in *arg2* (cast to *uid_t*). If the value −1 is specified, the caller's real user ID is used. The ID of the destination keyring is specified in *arg3* (cast to *key_serial_t*).

The caller must have the **CAP_SETUID** capability in its user namespace in order to fetch the persistent keyring for a user ID that does not match either the real or effective user ID of the caller.

If the call is successful, a link to the persistent keyring is added to the keyring whose ID was specified in *arg3*.

The caller must have *write* permission on the keyring.

The persistent keyring will be created by the kernel if it does not yet exist.

Each time the **KEYCTL_GET_PERSISTENT** operation is performed, the persistent keyring will have its expiration timeout reset to the value in:

```
/proc/sys/kernel/keys/persistent_keyring_expiry
```

Should the timeout be reached, the persistent keyring will be removed and everything it pins can then be garbage collected.

Persistent keyrings were added to Linux in kernel version 3.13.

The arguments *arg4* and *arg5* are ignored.

This operation is exposed by *libkeyutils* via the function **keyctl_get_persistent**(3).

KEYCTL_DH_COMPUTE (since Linux 4.7)

Compute a Diffie-Hellman shared secret or public key, optionally applying key derivation function (KDF) to the result.

The *arg2* argument is a pointer to a set of parameters containing serial numbers for three *"user"* keys used in the Diffie-Hellman calculation, packaged in a structure of the following form:

```
struct keyctl_dh_params {
    int32_t private; /* The local private key */
    int32_t prime; /* The prime, known to both parties */
    int32_t base;  /* The base integer: either a shared
                      generator or the remote public key */
};
```

Each of the three keys specified in this structure must grant the caller *read* permission. The payloads of these keys are used to calculate the Diffie-Hellman result as:

base ˆ 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.

The *arg3* argument (cast to *char* *) points to a buffer where the result of the calculation is placed. The size of that buffer is specified in *arg4* (cast to *size_t*).

The buffer must be large enough to accommodate the output data, otherwise an error is returned. If *arg4* is specified zero, in which case the buffer is not used and the operation returns the minimum required buffer size (i.e., the length of the prime).

Diffie-Hellman computations can be performed in user space, but require a multiple-precision integer (MPI) library. Moving the implementation into the kernel gives access to the kernel MPI implementation, and allows access to secure or acceleration hardware.

Adding support for DH computation to the **keyctl**() system call was considered a good fit due to the DH algorithm's use for deriving shared keys; it also allows the type of the key to determine which DH implementation (software or hardware) is appropriate.

If the *arg5* argument is **NULL**, then the DH result itself is returned. Otherwise (since Linux 4.12), it is a pointer to a structure which specifies parameters of the KDF operation to be applied:

```
struct keyctl_kdf_params {
    char *hashname;      /* Hash algorithm name */
    char *otherinfo;     /* SP800-56A OtherInfo */
    __u32 otherinfolen; /* Length of otherinfo data */
    __u32 __spare[8];   /* Reserved */
};
```

The *hashname* field is a null-terminated string which specifies a hash name (available in the kernel's crypto API; the list of the hashes available is rather tricky to observe; please refer to the "Kernel Crypto API Architecture" ⟨https://www.kernel.org/doc/html/latest/crypto/architecture.html⟩ documentation for the information regarding how hash names are constructed and your kernel's source and configuration regarding what ciphers and templates with type **CRYPTO_ALG_TYPE_SHASH** are available) to be applied to DH result in KDF operation.

The *otherinfo* field is an *OtherInfo* data as described in SP800-56A section 5.8.1.2 and is algorithm-specific. This data is concatenated with the result of DH operation and is provided as an input to the KDF operation. Its size is provided in the *otherinfolen* field and is limited by **KEYCTL_KDF_MAX_OI_LEN** constant that defined in *security/keys/internal.h* to a value of 64.

The **__spare** field is currently unused. It was ignored until Linux 4.13 (but still should be user-addressable since it is copied to the kernel), and should contain zeros since Linux 4.13.

The KDF implementation complies with SP800-56A as well as with SP800-108 (the counter KDF).

This operation is exposed by *libkeyutils* (from version 1.5.10 onwards) via the functions **keyctl_dh_compute**(3) and **keyctl_dh_compute_alloc**(3).

KEYCTL_RESTRICT_KEYRING (since Linux 4.12)

Apply a key-linking restriction to the keyring with the ID provided in *arg2* (cast to *key_serial_t*). The caller must have *setattr* permission on the key. If *arg3* is NULL, any attempt to add a key to the keyring is blocked; otherwise it contains a pointer to a string with a key type name and *arg4* contains a pointer to string that describes the type-specific restriction. As of Linux 4.12, only the type "asymmetric" has restrictions defined:

builtin_trusted

Allows only keys that are signed by a key linked to the built-in keyring (".builtin_trusted_keys").

builtin_and_secondary_trusted

Allows only keys that are signed by a key linked to the secondary keyring (".secondary_trusted_keys") or, by extension, a key in a built-in keyring, as the latter is linked to the former.

key_or_keyring:*key*
key_or_keyring:*key***:chain**

If *key* specifies the ID of a key of type "asymmetric", then only keys that are signed by this key are allowed.

If *key* specifies the ID of a keyring, then only keys that are signed by a key linked to this keyring are allowed.

If ":chain" is specified, keys that are signed by a keys linked to the destination keyring (that is, the keyring with the ID specified in the *arg2* argument) are also allowed.

Note that a restriction can be configured only once for the specified keyring; once a restriction is set, it can't be overridden.

The argument *arg5* is ignored.

RETURN VALUE

For a successful call, the return value depends on the operation:

KEYCTL_GET_KEYRING_ID
The ID of the requested keyring.

KEYCTL_JOIN_SESSION_KEYRING
The ID of the joined session keyring.

KEYCTL_DESCRIBE
The size of the description (including the terminating null byte), irrespective of the provided buffer size.

KEYCTL_SEARCH
The ID of the key that was found.

KEYCTL_READ
The amount of data that is available in the key, irrespective of the provided buffer size.

KEYCTL_SET_REQKEY_KEYRING
The ID of the previous default keyring to which implicitly requested keys were linked (one of **KEY_REQKEY_DEFL_USER_***).

KEYCTL_ASSUME_AUTHORITY
Either 0, if the ID given was 0, or the ID of the authorization key matching the specified key, if a nonzero key ID was provided.

KEYCTL_GET_SECURITY
The size of the LSM security label string (including the terminating null byte), irrespective of the provided buffer size.

KEYCTL_GET_PERSISTENT
The ID of the persistent keyring.

KEYCTL_DH_COMPUTE
The number of bytes copied to the buffer, or, if *arg4* is 0, the required buffer size.

All other operations
Zero.

On error, −1 is returned, and *errno* is set appropriately to indicate the error.

ERRORS

EACCES
The requested operation wasn't permitted.

EAGAIN
operation was **KEYCTL_DH_COMPUTE** and there was an error during crypto module initialization.

EDEADLK
operation was **KEYCTL_LINK** and the requested link would result in a cycle.

EDEADLK
operation was **KEYCTL_RESTRICT_KEYRING** and the requested keyring restriction would result in a cycle.

EDQUOT
The key quota for the caller's user would be exceeded by creating a key or linking it to the keyring.

EEXIST
operation was **KEYCTL_RESTRICT_KEYRING** and keyring provided in *arg2* argument already has a restriction set.

EFAULT
operation was **KEYCTL_DH_COMPUTE** and one of the following has failed:

- copying of the *struct keyctl_dh_params*, provided in the *arg2* argument, from user space;
- copying of the *struct keyctl_kdf_params*, provided in the non-NULL *arg5* argument, from user space (in case kernel supports performing KDF operation on DH operation result);
- copying of data pointed by the *hashname* field of the *struct keyctl_kdf_params* from user space;
- copying of data pointed by the *otherinfo* field of the *struct keyctl_kdf_params* from user space if the *otherinfolen* field was nonzero;
- copying of the result to user space.

EINVAL
operation was **KEYCTL_SETPERM** and an invalid permission bit was specified in *arg3*.

EINVAL
operation was **KEYCTL_SEARCH** and the size of the description in *arg4* (including the terminating null byte) exceeded 4096 bytes. size of the string (including the terminating null byte) specified in *arg3* (the key type) or *arg4* (the key description) exceeded the limit (32 bytes and 4096 bytes respectively).

EINVAL (Linux kernels before 4.12)
operation was **KEYCTL_DH_COMPUTE**, argument *arg5* was non-NULL.

EINVAL
operation was **KEYCTL_DH_COMPUTE** And the digest size of the hashing algorithm supplied is zero.

EINVAL
operation was **KEYCTL_DH_COMPUTE** and the buffer size provided is not enough to hold the result. Provide 0 as a buffer size in order to obtain the minimum buffer size.

EINVAL
operation was **KEYCTL_DH_COMPUTE** and the hash name provided in the *hashname* field of the *struct keyctl_kdf_params* pointed by *arg5* argument is too big (the limit is implementation-specific and varies between kernel versions, but it is deemed big enough for all valid algorithm names).

EINVAL
operation was **KEYCTL_DH_COMPUTE** and the *__spare* field of the *struct keyctl_kdf_params* provided in the *arg5* argument contains nonzero values.

EKEYEXPIRED
An expired key was found or specified.

EKEYREJECTED
A rejected key was found or specified.

EKEYREVOKED
A revoked key was found or specified.

ELOOP
operation was **KEYCTL_LINK** and the requested link would cause the maximum nesting depth for keyrings to be exceeded.

EMSGSIZE
operation was **KEYCTL_DH_COMPUTE** and the buffer length exceeds **KEYCTL_KDF_MAX_OUTPUT_LEN** (which is 1024 currently) or the *otherinfolen* field of the *struct keyctl_kdf_parms* passed in *arg5* exceeds **KEYCTL_KDF_MAX_OI_LEN** (which is 64 currently).

ENFILE (Linux kernels before 3.13)
operation was **KEYCTL_LINK** and the keyring is full. (Before Linux 3.13, the available space for storing keyring links was limited to a single page of memory; since Linux 3.13, there is no fixed limit.)

ENOENT
operation was **KEYCTL_UNLINK** and the key to be unlinked isn't linked to the keyring.

ENOENT
operation was **KEYCTL_DH_COMPUTE** and the hashing algorithm specified in the *hashname* field of the *struct keyctl_kdf_params* pointed by *arg5* argument hasn't been found.

ENOENT
operation was **KEYCTL_RESTRICT_KEYRING** and the type provided in *arg3* argument doesn't support setting key linking restrictions.

ENOKEY
No matching key was found or an invalid key was specified.

ENOKEY
The value **KEYCTL_GET_KEYRING_ID** was specified in *operation*, the key specified in *arg2* did not exist, and *arg3* was zero (meaning don't create the key if it didn't exist).

ENOMEM
One of kernel memory allocation routines failed during the execution of the syscall.

ENOTDIR
A key of keyring type was expected but the ID of a key with a different type was provided.

EOPNOTSUPP
operation was **KEYCTL_READ** and the key type does not support reading (e.g., the type is *"login"*).

EOPNOTSUPP
operation was **KEYCTL_UPDATE** and the key type does not support updating.

EOPNOTSUPP
operation was **KEYCTL_RESTRICT_KEYRING**, the type provided in *arg3* argument was "asymmetric", and the key specified in the restriction specification provided in *arg4* has type other than "asymmetric" or "keyring".

EPERM
operation was **KEYCTL_GET_PERSISTENT**, *arg2* specified a UID other than the calling thread's real or effective UID, and the caller did not have the **CAP_SETUID** capability.

EPERM
operation was **KEYCTL_SESSION_TO_PARENT** and either: all of the UIDs (GIDs) of the parent process do not match the effective UID (GID) of the calling process; the UID of the parent's existing session keyring or the UID of the caller's session keyring did not match the effective UID of the caller; the parent process is not single-thread; or the parent process is **init**(1) or a kernel

thread.

ETIMEDOUT
operation was **KEYCTL_DH_COMPUTE** and the initialization of crypto modules has timed out.

VERSIONS

This system call first appeared in Linux 2.6.10.

CONFORMING TO

This system call is a nonstandard Linux extension.

NOTES

No wrapper for this system call is provided in glibc. A wrapper is provided in the *libkeyutils* library. When employing the wrapper in that library, link with *−lkeyutils*. However, rather than using this system call directly, you probably want to use the various library functions mentioned in the descriptions of individual operations above.

EXAMPLE

The program below provide subset of the functionality of the **request-key**(8) program provided by the *keyutils* package. For informational purposes, the program records various information in a log file.

As described in **request_key**(2), the **request-key**(8) program is invoked with command-line arguments that describe a key that is to be instantiated. The example program fetches and logs these arguments. The program assumes authority to instantiate the requested key, and then instantiates that key.

The following shell session demonstrates the use of this program. In the session, we compile the program and then use it to temporarily replace the standard **request-key**(8) program. (Note that temporarily disabling the standard **request-key**(8) program may not be safe on some systems.) While our example program is installed, we use the example program shown in **request_key**(2) to request a key.

```
$ cc -o key_instantiate key_instantiate.c -lkeyutils
$ sudo mv /sbin/request-key /sbin/request-key.backup
$ sudo cp key_instantiate /sbin/request-key
$ ./t_request_key user mykey somepayloaddata
Key ID is 20d035bf
$ sudo mv /sbin/request-key.backup /sbin/request-key
```

Looking at the log file created by this program, we can see the command-line arguments supplied to our example program:

```
$ cat /tmp/key_instantiate.log
Time: Mon Nov  7 13:06:47 2016

Command line arguments:
  argv[0]:            /sbin/request-key
  operation:          create
  key_to_instantiate: 20d035bf
  UID:                1000
  GID:                1000
  thread_keyring:     0
  process_keyring:    0
  session_keyring:    256e6a6

Key description:      user;1000;1000;3f010000;mykey
Auth key payload:     somepayloaddata
Destination keyring:  256e6a6
Auth key description: .request_key_auth;1000;1000;0b010000;20d035bf
```

The last few lines of the above output show that the example program was able to fetch:

* the description of the key to be instantiated, which included the name of the key (*mykey*);
* the payload of the authorization key, which consisted of the data (*somepayloaddata*) passed to **request_key**(2);
* the destination keyring that was specified in the call to **request_key**(2); and
* the description of the authorization key, where we can see that the name of the authorization key matches the ID of the key that is to be instantiated (*20d035bf*).

The example program in **request_key**(2) specified the destination keyring as **KEY_SPEC_SESSION_KEYRING**. By examining the contents of */proc/keys*, we can see that this was translated to the ID of the destination keyring (*0256e6a6*) shown in the log output above; we can also see the newly created key with the name *mykey* and ID *20d035bf*.

```
$ cat /proc/keys | egrep 'mykey|256e6a6'
0256e6a6 I--Q---  194 perm 3f030000  1000  1000 keyring  _ses: 3
20d035bf I--Q---    1 perm 3f010000  1000  1000 user     mykey: 16
```

Program source

```
/* key_instantiate.c */

#include <sys/types.h>
#include <keyutils.h>
#include <time.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#ifndef KEY_SPEC_REQUESTOR_KEYRING
#define KEY_SPEC_REQUESTOR_KEYRING      -8
#endif

int
main(int argc, char *argv[])
{
    FILE *fp;
    time_t t;
    char *operation;
    key_serial_t key_to_instantiate, dest_keyring;
    key_serial_t thread_keyring, process_keyring, session_keyring;
    uid_t uid;
    gid_t gid;
    char dbuf[256];
    char auth_key_payload[256];
    int akp_size;       /* Size of auth_key_payload */

    fp = fopen("/tmp/key_instantiate.log", "w");
    if (fp == NULL)
        exit(EXIT_FAILURE);

    setbuf(fp, NULL);

    t = time(NULL);
    fprintf(fp, "Time: %s\n", ctime(&t));

    /*
     * The kernel passes a fixed set of arguments to the program
     * that it execs; fetch them.
     */
    operation = argv[1];
    key_to_instantiate = atoi(argv[2]);
    uid = atoi(argv[3]);
    gid = atoi(argv[4]);
    thread_keyring = atoi(argv[5]);
    process_keyring = atoi(argv[6]);
    session_keyring = atoi(argv[7]);

    fprintf(fp, "Command line arguments:\n");
    fprintf(fp, "  argv[0]:            %s\n", argv[0]);
    fprintf(fp, "  operation:          %s\n", operation);
    fprintf(fp, "  key_to_instantiate: %lx\n",
            (long) key_to_instantiate);
    fprintf(fp, "  UID:                %ld\n", (long) uid);
    fprintf(fp, "  GID:                %ld\n", (long) gid);
    fprintf(fp, "  thread_keyring:     %lx\n", (long) thread_keyring);
    fprintf(fp, "  process_keyring:    %lx\n", (long) process_keyring);
    fprintf(fp, "  session_keyring:    %lx\n", (long) session_keyring);
    fprintf(fp, "\n");

    /*
     * Assume the authority to instantiate the key named in argv[2]
     */
    if (keyctl(KEYCTL_ASSUME_AUTHORITY, key_to_instantiate) == -1) {
        fprintf(fp, "KEYCTL_ASSUME_AUTHORITY failed: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    /*
     * Fetch the description of the key that is to be instantiated
     */
    if (keyctl(KEYCTL_DESCRIBE, key_to_instantiate,
                dbuf, sizeof(dbuf)) == -1) {
        fprintf(fp, "KEYCTL_DESCRIBE failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    fprintf(fp, "Key description:      %s\n", dbuf);

    /*
     * Fetch the payload of the authorization key, which is
     * actually the callout data given to request_key()
     */
    akp_size = keyctl(KEYCTL_READ, KEY_SPEC_REQKEY_AUTH_KEY,
                      auth_key_payload, sizeof(auth_key_payload));
    if (akp_size == -1) {
        fprintf(fp, "KEYCTL_READ failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    auth_key_payload[akp_size] = '\0';
    fprintf(fp, "Auth key payload:     %s\n", auth_key_payload);

    /*
     * For interest, get the ID of the authorization key and
     * display it.
     */
    auth_key = keyctl(KEYCTL_GET_KEYRING_ID,
            KEY_SPEC_REQKEY_AUTH_KEY);
    if (auth_key == -1) {
        fprintf(fp, "KEYCTL_GET_KEYRING_ID failed: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    fprintf(fp, "Auth key ID:          %lx\n", (long) auth_key);

    /*
     * Fetch key ID for the request_key(2) destination keyring.
     */
    dest_keyring = keyctl(KEYCTL_GET_KEYRING_ID,
                          KEY_SPEC_REQUESTOR_KEYRING);
    if (dest_keyring == -1) {
        fprintf(fp, "KEYCTL_GET_KEYRING_ID failed: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    fprintf(fp, "Destination keyring:  %lx\n", (long) dest_keyring);

    /*
     * Fetch the description of the authorization key. This
     * allows us to see the key type, UID, GID, permissions,
     * and description (name) of the key. Among other things,
     * we will see that the name of the key is a hexadecimal
     * string representing the ID of the key to be instantiated.
     */
    if (keyctl(KEYCTL_DESCRIBE, KEY_SPEC_REQKEY_AUTH_KEY,
                dbuf, sizeof(dbuf)) == -1) {
        fprintf(fp, "KEYCTL_DESCRIBE failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    fprintf(fp, "Auth key description: %s\n", dbuf);

    /*
     * Instantiate the key using the callout data that was supplied
     * in the payload of the authorization key.
     */
    if (keyctl(KEYCTL_INSTANTIATE, key_to_instantiate,
               auth_key_payload, akp_size + 1, dest_keyring) == -1) {
```

```
            fprintf(fp, "KEYCTL_INSTANTIATE failed: %s\n",
                    strerror(errno));
            exit(EXIT_FAILURE);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

keyctl(1), **add_key**(2), **request_key**(2), **keyctl**(3), **keyctl_assume_authority**(3), **keyctl_chown**(3), **keyctl_clear**(3), **keyctl_describe**(3), **keyctl_describe_alloc**(3), **keyctl_dh_compute**(3), **keyctl_dh_compute_alloc**(3), **keyctl_get_keyring_ID**(3), **keyctl_get_persistent**(3), **keyctl_get_security**(3), **keyctl_get_security_alloc**(3), **keyctl_instantiate**(3), **keyctl_instantiate_iov**(3), **keyctl_invalidate**(3), **keyctl_join_session_keyring**(3), **keyctl_link**(3), **keyctl_negate**(3), **keyctl_read**(3), **keyctl_read_alloc**(3), **keyctl_reject**(3), **keyctl_revoke**(3), **keyctl_search**(3), **keyctl_session_to_parent**(3), **keyctl_set_reqkey_keyring**(3), **keyctl_set_timeout**(3), **keyctl_setperm**(3), **keyctl_unlink**(3), **keyctl_update**(3), **recursive_key_scan**(3), **recursive_session_key_scan**(3), **capabilities**(7), **credentials**(7), **keyrings**(7), **keyutils**(7), **persistent−keyring**(7), **process−keyring**(7), **session−keyring**(7), **thread−keyring**(7), **user−keyring**(7), **user_namespaces**(7), **user−session−keyring**(7), **request−key**(8)

The kernel source files under *Documentation/security/keys/* (or, before Linux 4.13, in the file *Documentation/security/keys.txt*).

COLOPHON

This page is part of release 5.05 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.