NAME

keyrings − in-kernel key management and retention facility

DESCRIPTION

The Linux key-management facility is primarily a way for various kernel components to retain or cache security data, authentication keys, encryption keys, and other data in the kernel.

System call interfaces are provided so that user-space programs can manage those objects and also use the facility for their own purposes; see **add_key**(2), **request_key**(2), and **keyctl**(2).

A library and some user-space utilities are provided to allow access to the facility. See **keyctl**(1), **keyctl**(3), and **keyutils**(7) for more information.

Keys

A key has the following attributes:

Serial number (ID)
: This is a unique integer handle by which a key is referred to in system calls. The serial number is sometimes synonymously referred as the key ID. Programmatically, key serial numbers are represented using the type *key_serial_t*.

Type
: A key's type defines what sort of data can be held in the key, how the proposed content of the key will be parsed, and how the payload will be used.

There are a number of general-purpose types available, plus some specialist types defined by specific kernel components.

Description (name)
: The key description is a printable string that is used as the search term for the key (in conjunction with the key type) as well as a display name. During searches, the description may be partially matched or exactly matched.

Payload (data)
: The payload is the actual content of a key. This is usually set when a key is created, but it is possible for the kernel to upcall to user space to finish the instantiation of a key if that key wasn't already known to the kernel when it was requested. For further details, see **request_key**(2).

A key's payload can be read and updated if the key type supports it and if suitable permission is granted to the caller.

Access rights
: Much as files do, each key has an owning user ID, an owning group ID, and a security label. Each key also has a set of permissions, though there are more than for a normal UNIX file, and there is an additional category—possessor—beyond the usual user, group, and other (see *Possession*, below).

Note that keys are quota controlled, since they require unswappable kernel memory. The owning user ID specifies whose quota is to be debited.

Expiration time
: Each key can have an expiration time set. When that time is reached, the key is marked as being expired and accesses to it fail with the error **EKEYEXPIRED**. If not deleted, updated, or replaced, then, after a set amount of time, an expired key is automatically removed (garbage collected) along with all links to it, and attempts to access the key fail with the error **ENOKEY**.

Reference count
: Each key has a reference count. Keys are referenced by keyrings, by currently active users, and by a process's credentials. When the reference count reaches zero, the key is scheduled for garbage collection.

Key types

The kernel provides several basic types of key:

"keyring"
Keyrings are special keys which store a set of links to other keys (including other keyrings), analogous to a directory holding links to files. The main purpose of a keyring is to prevent other keys from being garbage collected because nothing refers to them.

Keyrings with descriptions (names) that begin with a period ('.') are reserved to the implementation.

"user"
This is a general-purpose key type. The key is kept entirely within kernel memory. The payload may be read and updated by user-space applications.

The payload for keys of this type is a blob of arbitrary data of up to 32,767 bytes.

The description may be any valid string, though it is preferred that it start with a colon-delimited prefix representing the service to which the key is of interest (for instance *"afs:mykey"*).

"logon" (since Linux 3.3)
This key type is essentially the same as *"user"*, but it does not provide reading (i.e., the **keyctl**(2) **KEYCTL_READ** operation), meaning that the key payload is never visible from user space. This is suitable for storing username-password pairs that should not be readable from user space.

The description of a *"logon"* key *must* start with a non-empty colon-delimited prefix whose purpose is to identify the service to which the key belongs. (Note that this differs from keys of the *"user"* type, where the inclusion of a prefix is recommended but is not enforced.)

"big_key" (since Linux 3.13)
This key type is similar to the *"user"* key type, but it may hold a payload of up to 1 MiB in size. This key type is useful for purposes such as holding Kerberos ticket caches.

The payload data may be stored in a tmpfs filesystem, rather than in kernel memory, if the data size exceeds the overhead of storing the data in the filesystem. (Storing the data in a filesystem requires filesystem structures to be allocated in the kernel. The size of these structures determines the size threshold above which the tmpfs storage method is used.) Since Linux 4.8, the payload data is encrypted when stored in tmpfs, thereby preventing it from being written unencrypted into swap space.

There are more specialized key types available also, but they aren't discussed here because they aren't intended for normal user-space use.

Key type names that begin with a period ('.') are reserved to the implementation.

Keyrings

As previously mentioned, keyrings are a special type of key that contain links to other keys (which may include other keyrings). Keys may be linked to by multiple keyrings. Keyrings may be considered as analogous to UNIX directories where each directory contains a set of hard links to files.

Various operations (system calls) may be applied only to keyrings:

Adding
A key may be added to a keyring by system calls that create keys. This prevents the new key from being immediately deleted when the system call releases its last reference to the key.

Linking
A link may be added to a keyring pointing to a key that is already known, provided this does not create a self-referential cycle.

Unlinking
A link may be removed from a keyring. When the last link to a key is removed, that key will be scheduled for deletion by the garbage collector.

Clearing
All the links may be removed from a keyring.

Searching
A keyring may be considered the root of a tree or subtree in which keyrings form the branches and non-keyrings the leaves. This tree may be searched for a key matching a particular type and

description.

See **keyctl_clear**(3), **keyctl_link**(3), **keyctl_search**(3), and **keyctl_unlink**(3) for more information.

Anchoring keys

To prevent a key from being garbage collected, it must be anchored to keep its reference count elevated when it is not in active use by the kernel.

Keyrings are used to anchor other keys: each link is a reference on a key. Note that keyrings themselves are just keys and are also subject to the same anchoring requirement to prevent them being garbage collected.

The kernel makes available a number of anchor keyrings. Note that some of these keyrings will be created only when first accessed.

Process keyrings

Process credentials themselves reference keyrings with specific semantics. These keyrings are pinned as long as the set of credentials exists, which is usually as long as the process exists.

There are three keyrings with different inheritance/sharing rules: the **session-keyring**(7) (inherited and shared by all child processes), the **process-keyring**(7) (shared by all threads in a process) and the **thread-keyring**(7) (specific to a particular thread).

As an alternative to using the actual keyring IDs, in calls to **add_key**(2), **keyctl**(2), and **request_key**(2), the special keyring values **KEY_SPEC_SESSION_KEYRING**, **KEY_SPEC_PROCESS_KEYRING**, and **KEY_SPEC_THREAD_KEYRING** can be used to refer to the caller's own instances of these keyrings.

User keyrings

Each UID known to the kernel has a record that contains two keyrings: the **user-keyring**(7) and the **user-session-keyring**(7). These exist for as long as the UID record in the kernel exists.

As an alternative to using the actual keyring IDs, in calls to **add_key**(2), **keyctl**(2), and **request_key**(2), the special keyring values **KEY_SPEC_USER_KEYRING** and **KEY_SPEC_USER_SESSION_KEYRING** can be used to refer to the caller's own instances of these keyrings.

A link to the user keyring is placed in a new session keyring by **pam_keyinit**(8) when a new login session is initiated.

Persistent keyrings

There is a **persistent-keyring**(7) available to each UID known to the system. It may persist beyond the life of the UID record previously mentioned, but has an expiration time set such that it is automatically cleaned up after a set time. The persistent keyring permits, for example, **cron**(8) scripts to use credentials that are left in the persistent keyring after the user logs out.

Note that the expiration time of the persistent keyring is reset every time the persistent key is requested.

Special keyrings

There are special keyrings owned by the kernel that can anchor keys for special purposes. An example of this is the *system keyring* used for holding encryption keys for module signature verification.

These special keyrings are usually closed to direct alteration by user space.

An originally planned "group keyring", for storing keys associated with each GID known to the kernel, is not so far implemented, is unlikely to be implemented. Nevertheless, the constant **KEY_SPEC_GROUP_KEYRING** has been defined for this keyring.

Possession

The concept of possession is important to understanding the keyrings security model. Whether a thread possesses a key is determined by the following rules:

(1) Any key or keyring that does not grant *search* permission to the caller is ignored in all the following rules.

(2) A thread possesses its **session-keyring**(7), **process-keyring**(7), and **thread-keyring**(7) directly because those keyrings are referred to by its credentials.

(3) If a keyring is possessed, then any key it links to is also possessed.

(4) If any key a keyring links to is itself a keyring, then rule (3) applies recursively.

(5) If a process is upcalled from the kernel to instantiate a key (see **request_key**(2)), then it also possesses the requester's keyrings as in rule (1) as if it were the requester.

Note that possession is not a fundamental property of a key, but must rather be calculated each time the key is needed.

Possession is designed to allow set-user-ID programs run from, say a user's shell to access the user's keys. Granting permissions to the key possessor while denying them to the key owner and group allows the prevention of access to keys on the basis of UID and GID matches.

When it creates the session keyring, **pam_keyinit**(8) adds a link to the **user-keyring**(7), thus making the user keyring and anything it contains possessed by default.

Access rights

Each key has the following security-related attributes:

* The owning user ID
* The ID of a group that is permitted to access the key
* A security label
* A permissions mask

The permissions mask contains four sets of rights. The first three sets are mutually exclusive. One and only one will be in force for a particular access check. In order of descending priority, these three sets are:

user The set specifies the rights granted if the key's user ID matches the caller's filesystem user ID.

group The set specifies the rights granted if the user ID didn't match and the key's group ID matches the caller's filesystem GID or one of the caller's supplementary group IDs.

other The set specifies the rights granted if neither the key's user ID nor group ID matched.

The fourth set of rights is:

possessor
The set specifies the rights granted if a key is determined to be possessed by the caller.

The complete set of rights for a key is the union of whichever of the first three sets is applicable plus the fourth set if the key is possessed.

The set of rights that may be granted in each of the four masks is as follows:

view The attributes of the key may be read. This includes the type, description, and access rights (excluding the security label).

read For a key: the payload of the key may be read. For a keyring: the list of serial numbers (keys) to which the keyring has links may be read.

write The payload of the key may be updated and the key may be revoked. For a keyring, links may be added to or removed from the keyring, and the keyring may be cleared completely (all links are removed),

search For a key (or a keyring): the key may be found by a search. For a keyring: keys and keyrings that are linked to by the keyring may be searched.

link Links may be created from keyrings to the key. The initial link to a key that is established when the key is created doesn't require this permission.

setattr The ownership details and security label of the key may be changed, the key's expiration time may be set, and the key may be revoked.

In addition to access rights, any active Linux Security Module (LSM) may prevent access to a key if its policy so dictates. A key may be given a security label or other attribute by the LSM; this label is retrievable via **keyctl_get_security**(3).

See **keyctl_chown**(3), **keyctl_describe**(3), **keyctl_get_security**(3), **keyctl_setperm**(3), and **selinux**(8) for more information.

Searching for keys

One of the key features of the Linux key-management facility is the ability to find a key that a process is retaining. The **request_key**(2) system call is the primary point of access for user-space applications to find a key. (Internally, the kernel has something similar available for use by internal components that make use of keys.)

The search algorithm works as follows:

(1) The process keyrings are searched in the following order: the thread **thread-keyring**(7) if it exists, the **process-keyring**(7) if it exists, and then either the **session-keyring**(7) if it exists or the **user-session-keyring**(7) if that exists.

(2) If the caller was a process that was invoked by the **request_key**(2) upcall mechanism, then the keyrings of the original caller of **request_key**(2) will be searched as well.

(3) The search of a keyring tree is in breadth-first order: each keyring is searched first for a match, then the keyrings referred to by that keyring are searched.

(4) If a matching key is found that is valid, then the search terminates and that key is returned.

(5) If a matching key is found that has an error state attached, that error state is noted and the search continues.

(6) If no valid matching key is found, then the first noted error state is returned; otherwise, an **ENOKEY** error is returned.

It is also possible to search a specific keyring, in which case only steps (3) to (6) apply.

See **request_key**(2) and **keyctl_search**(3) for more information.

On-demand key creation

If a key cannot be found, **request_key**(2) will, if given a *callout_info* argument, create a new key and then upcall to user space to instantiate the key. This allows keys to be created on an as-needed basis.

Typically, this will involve the kernel creating a new process that executes the **request-key**(8) program, which will then execute the appropriate handler based on its configuration.

The handler is passed a special authorization key that allows it and only it to instantiate the new key. This is also used to permit searches performed by the handler program to also search the requester's keyrings.

See **request_key**(2), **keyctl_assume_authority**(3), **keyctl_instantiate**(3), **keyctl_negate**(3), **keyctl_reject**(3), **request-key**(8), and **request-key.conf**(5) for more information.

/proc files

The kernel provides various */proc* files that expose information about keys or define limits on key usage.

/proc/keys (since Linux 2.6.10)

This file exposes a list of the keys for which the reading thread has *view* permission, providing various information about each key. The thread need not possess the key for it to be visible in this file.

The only keys included in the list are those that grant *view* permission to the reading process (regardless of whether or not it possesses them). LSM security checks are still performed, and may filter out further keys that the process is not authorized to view.

An example of the data that one might see in this file (with the columns numbered for easy reference below) is the following:

```
  (1)     (2)     (3)(4)    (5)     (6)   (7)   (8)        (9)
009a2028 I--Q---   1 perm 3f010000  1000  1000 user     krb_ccache:primary: 12
1806c4ba I--Q---   1 perm 3f010000  1000  1000 keyring  _pid: 2
25d3a08f I--Q---   1 perm 1f3f0000  1000 65534 keyring  _uid_ses.1000: 1
28576bd8 I--Q---   3 perm 3f010000  1000  1000 keyring  _krb: 1
2c546d21 I--Q--- 190 perm 3f030000  1000  1000 keyring  _ses: 2
30a4e0be I------   4   2d 1f030000  1000 65534 keyring  _persistent.1000: 1
32100fab I--Q---   4 perm 1f3f0000  1000 65534 keyring  _uid.1000: 2
32a387ea I--Q---   1 perm 3f010000  1000  1000 keyring  _pid: 2
3ce56aea I--Q---   5 perm 3f030000  1000  1000 keyring  _ses: 1
```

The fields shown in each line of this file are as follows:

ID (1) The ID (serial number) of the key, expressed in hexadecimal.

Flags (2)
A set of flags describing the state of the key:

I The key has been instantiated.

R The key has been revoked.

D The key is dead (i.e., the key type has been unregistered). (A key may be briefly in this state during garbage collection.)

Q The key contributes to the user's quota.

U The key is under construction via a callback to user space; see **request-key**(2).

N The key is negatively instantiated.

i The key has been invalidated.

Usage (3)
This is a count of the number of kernel credential structures that are pinning the key (approximately: the number of threads and open file references that refer to this key).

Timeout (4)
The amount of time until the key will expire, expressed in human-readable form (weeks, days, hours, minutes, and seconds). The string *perm* here means that the key is permanent (no timeout). The string *expd* means that the key has already expired, but has not yet been garbage collected.

Permissions (5)
The key permissions, expressed as four hexadecimal bytes containing, from left to right, the possessor, user, group, and other permissions. Within each byte, the permission bits are as follows:

0x01	*view*
Ox02	*read*
0x04	*write*
0x08	*search*
0x10	*link*
0x20	*setattr*

UID (6)
The user ID of the key owner.

GID (7)
The group ID of the key. The value −1 here means that the key has no group ID; this can occur in certain circumstances for keys created by the kernel.

Type (8)
The key type (user, keyring, etc.)

Description (9)

The key description (name). This field contains descriptive information about the key. For most key types, it has the form

name[: extra−info]

The *name* subfield is the key's description (name). The optional *extra−info* field provides some further information about the key. The information that appears here depends on the key type, as follows:

"user" and *"logon"*

The size in bytes of the key payload (expressed in decimal).

"keyring"

The number of keys linked to the keyring, or the string *empty* if there are no keys linked to the keyring.

"big_key"

The payload size in bytes, followed either by the string *[file]*, if the key payload exceeds the threshold that means that the payload is stored in a (swappable) **tmpfs**(5) filesystem, or otherwise the string *[buff]*, indicating that the key is small enough to reside in kernel memory.

For the *".request_key_auth"* key type (authorization key; see **request_key**(2)), the description field has the form shown in the following example:

key:c9a9b19 pid:28880 ci:10

The three subfields are as follows:

key The hexadecimal ID of the key being instantiated in the requesting program.

pid The PID of the requesting program.

ci The length of the callout data with which the requested key should be instantiated (i.e., the length of the payload associated with the authorization key).

/proc/key-users (since Linux 2.6.10)

This file lists various information for each user ID that has at least one key on the system. An example of the data that one might see in this file is the following:

```
    0:     10 9/9 2/1000000 22/25000000
   42:      9 9/9 8/200 106/20000
 1000:     11 11/11 10/200 271/20000
```

The fields shown in each line are as follows:

uid The user ID.

usage This is a kernel-internal usage count for the kernel structure used to record key users.

nkeys/*nikeys*

The total number of keys owned by the user, and the number of those keys that have been instantiated.

qnkeys/*maxkeys*

The number of keys owned by the user, and the maximum number of keys that the user may own.

qnbytes/*maxbytes*

The number of bytes consumed in payloads of the keys owned by this user, and the upper limit on the number of bytes in key payloads for that user.

/proc/sys/kernel/keys/gc_delay (since Linux 2.6.32)

The value in this file specifies the interval, in seconds, after which revoked and expired keys will be garbage collected. The purpose of having such an interval is so that there is a window of time where user space can see an error (respectively **EKEYREVOKED** and **EKEYEXPIRED**) that indicates what happened to the key.

The default value in this file is 300 (i.e., 5 minutes).

/proc/sys/kernel/keys/persistent_keyring_expiry (since Linux 3.13)
: This file defines an interval, in seconds, to which the persistent keyring's expiration timer is reset each time the keyring is accessed (via **keyctl_get_persistent**(3) or the **keyctl**(2) **KEYCTL_GET_PERSISTENT** operation.)

The default value in this file is 259200 (i.e., 3 days).

The following files (which are writable by privileged processes) are used to enforce quotas on the number of keys and number of bytes of data that can be stored in key payloads:

/proc/sys/kernel/keys/maxbytes (since Linux 2.6.26)
: This is the maximum number of bytes of data that a nonroot user can hold in the payloads of the keys owned by the user.

The default value in this file is 20,000.

/proc/sys/kernel/keys/maxkeys (since Linux 2.6.26)
: This is the maximum number of keys that a nonroot user may own.

The default value in this file is 200.

/proc/sys/kernel/keys/root_maxbytes (since Linux 2.6.26)
: This is the maximum number of bytes of data that the root user (UID 0 in the root user namespace) can hold in the payloads of the keys owned by root.

The default value in this file is 25,000,000 (20,000 before Linux 3.17).

/proc/sys/kernel/keys/root_maxkeys (since Linux 2.6.26)
: This is the maximum number of keys that the root user (UID 0 in the root user namespace) may own.

The default value in this file is 1,000,000 (200 before Linux 3.17).

With respect to keyrings, note that each link in a keyring consumes 4 bytes of the keyring payload.

Users

The Linux key-management facility has a number of users and usages, but is not limited to those that already exist.

In-kernel users of this facility include:

Network filesystems - DNS
: The kernel uses the upcall mechanism provided by the keys to upcall to user space to do DNS lookups and then to cache the results.

AF_RXRPC and kAFS - Authentication
: The AF_RXRPC network protocol and the in-kernel AFS filesystem use keys to store the ticket needed to do secured or encrypted traffic. These are then looked up by network operations on AF_RXRPC and filesystem operations on kAFS.

NFS - User ID mapping
: The NFS filesystem uses keys to store mappings of foreign user IDs to local user IDs.

CIFS - Password
: The CIFS filesystem uses keys to store passwords for accessing remote shares.

Module verification
: The kernel build process can be made to cryptographically sign modules. That signature is then checked when a module is loaded.

User-space users of this facility include:

Kerberos key storage

The MIT Kerberos 5 facility (libkrb5) can use keys to store authentication tokens which can be made to be automatically cleaned up a set time after the user last uses them, but until then permits them to hang around after the user has logged out so that **cron**(8) scripts can use them.

SEE ALSO

keyctl(1), **add_key**(2), **keyctl**(2), **request_key**(2), **keyctl**(3), **keyutils**(7), **persistent−keyring**(7), **process−keyring**(7), **session−keyring**(7), **thread−keyring**(7), **user−keyring**(7), **user−session−keyring**(7), **pam_keyinit**(8), **request-key**(8)

The kernel source files *Documentation/crypto/asymmetric-keys.txt* and 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/.