Linux Security Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

CHAPTER

ONE

CREDENTIALS IN LINUX

By: David Howells <dhowells@redhat.com>

1.1 Overview

There are several parts to the security check performed by Linux when one object acts upon another:

1. Objects.

   Objects are things in the system that may be acted upon directly by userspace programs. Linux has a variety of actionable objects, including:

   - Tasks
   - Files/inodes
   - Sockets
   - Message queues
   - Shared memory segments
   - Semaphores

- Keys

As a part of the description of all these objects there is a set of credentials. What's in the set depends on the type of object.

2. Object ownership.

   Amongst the credentials of most objects, there will be a subset that indicates the ownership of that object. This is used for resource accounting and limitation (disk quotas and task rlimits for example).

   In a standard UNIX filesystem, for instance, this will be defined by the UID marked on the inode.

3. The objective context.

   Also amongst the credentials of those objects, there will be a subset that indicates the 'objective context' of that object. This may or may not be the same set as in (2) - in standard UNIX files, for instance, this is the defined by the UID and the GID marked on the inode.

   The objective context is used as part of the security calculation that is carried out when an object is acted upon.

4. Subjects.

   A subject is an object that is acting upon another object.

   Most of the objects in the system are inactive: they don't act on other objects within the system. Processes/tasks are the obvious exception: they do stuff; they access and manipulate things.

   Objects other than tasks may under some circumstances also be subjects. For instance an open file may send SIGIO to a task using the UID and EUID given to it by a task that called `fcntl(F_SETOWN)` upon it. In this case, the file struct will have a subjective context too.

5. The subjective context.

   A subject has an additional interpretation of its credentials. A subset of its credentials forms the 'subjective context'. The subjective context is used as part of the security calculation that is carried out when a subject acts.

   A Linux task, for example, has the FSUID, FSGID and the supplementary group list for when it is acting upon a file - which are quite separate from the real UID and GID that normally form the objective context of the task.

6. Actions.

   Linux has a number of actions available that a subject may perform upon an object. The set of actions available depends on the nature of the subject and the object.

   Actions include reading, writing, creating and deleting files; forking or signalling and tracing tasks.

7. Rules, access control lists and security calculations.

   When a subject acts upon an object, a security calculation is made. This involves taking the subjective context, the objective context and the action, and searching one or more sets of rules to see whether the subject is granted or denied permission to act in the desired manner on the object, given those contexts.

There are two main sources of rules:

a. Discretionary access control (DAC):

   Sometimes the object will include sets of rules as part of its description. This is an 'Access Control List' or 'ACL'. A Linux file may supply more than one ACL.

   A traditional UNIX file, for example, includes a permissions mask that is an abbreviated ACL with three fixed classes of subject ('user', 'group' and 'other'), each of which may be granted certain privileges ('read', 'write' and 'execute' - whatever those map to for the object in question). UNIX file permissions do not allow the arbitrary specification of subjects, however, and so are of limited use.

   A Linux file might also sport a POSIX ACL. This is a list of rules that grants various permissions to arbitrary subjects.

b. Mandatory access control (MAC):

   The system as a whole may have one or more sets of rules that get applied to all subjects and objects, regardless of their source. SELinux and Smack are examples of this.

   In the case of SELinux and Smack, each object is given a label as part of its credentials. When an action is requested, they take the subject label, the object label and the action and look for a rule that says that this action is either granted or denied.

1.2 Types of Credentials

The Linux kernel supports the following types of credentials:

1. Traditional UNIX credentials.

   - Real User ID
   - Real Group ID

   The UID and GID are carried by most, if not all, Linux objects, even if in some cases it has to be invented (FAT or CIFS files for example, which are derived from Windows). These (mostly) define the objective context of that object, with tasks being slightly different in some cases.

   - Effective, Saved and FS User ID
   - Effective, Saved and FS Group ID
   - Supplementary groups

   These are additional credentials used by tasks only. Usually, an EUID/EGID/GROUPS will be used as the subjective context, and real UID/GID will be used as the objective. For tasks, it should be noted that this is not always true.

2. Capabilities.

   - Set of permitted capabilities

- Set of inheritable capabilities
- Set of effective capabilities
- Capability bounding set

These are only carried by tasks. They indicate superior capabilities granted piecemeal to a task that an ordinary task wouldn't otherwise have. These are manipulated implicitly by changes to the traditional UNIX credentials, but can also be manipulated directly by the `capset()` system call.

The permitted capabilities are those caps that the process might grant itself to its effective or permitted sets through `capset()`. This inheritable set might also be so constrained.

The effective capabilities are the ones that a task is actually allowed to make use of itself.

The inheritable capabilities are the ones that may get passed across `execve()`.

The bounding set limits the capabilities that may be inherited across `execve()`, especially when a binary is executed that will execute as UID 0.

3. Secure management flags (securebits).

   These are only carried by tasks. These govern the way the above credentials are manipulated and inherited over certain operations such as execve(). They aren't used directly as objective or subjective credentials.

4. Keys and keyrings.

   These are only carried by tasks. They carry and cache security tokens that don't fit into the other standard UNIX credentials. They are for making such things as network filesystem keys available to the file accesses performed by processes, without the necessity of ordinary programs having to know about security details involved.

   Keyrings are a special type of key. They carry sets of other keys and can be searched for the desired key. Each process may subscribe to a number of keyrings:

   Per-thread keying Per-process keyring Per-session keyring

   When a process accesses a key, if not already present, it will normally be cached on one of these keyrings for future accesses to find.

   For more information on using keys, see `Documentation/security/keys/*`.

5. LSM

   The Linux Security Module allows extra controls to be placed over the operations that a task may do. Currently Linux supports several LSM options.

   Some work by labelling the objects in a system and then applying sets of rules (policies) that say what operations a task with one label may do to an object with another label.

6. AF_KEY

This is a socket-based approach to credential management for networking stacks [RFC 2367]. It isn't discussed by this document as it doesn't interact directly with task and file credentials; rather it keeps system level credentials.

When a file is opened, part of the opening task's subjective context is recorded in the file struct created. This allows operations using that file struct to use those credentials instead of the subjective context of the task that issued the operation. An example of this would be a file opened on a network filesystem where the credentials of the opened file should be presented to the server, regardless of who is actually doing a read or a write upon it.

1.3 File Markings

Files on disk or obtained over the network may have annotations that form the objective security context of that file. Depending on the type of filesystem, this may include one or more of the following:

- UNIX UID, GID, mode;
- Windows user ID;
- Access control list;
- LSM security label;
- UNIX exec privilege escalation bits (SUID/SGID);
- File capabilities exec privilege escalation bits.

These are compared to the task's subjective security context, and certain operations allowed or disallowed as a result. In the case of execve(), the privilege escalation bits come into play, and may allow the resulting process extra privileges, based on the annotations on the executable file.

1.4 Task Credentials

In Linux, all of a task's credentials are held in (uid, gid) or through (groups, keys, LSM security) a refcounted structure of type 'struct cred'. Each task points to its credentials by a pointer called 'cred' in its task_struct.

Once a set of credentials has been prepared and committed, it may not be changed, barring the following exceptions:

1. its reference count may be changed;
2. the reference count on the group_info struct it points to may be changed;
3. the reference count on the security data it points to may be changed;
4. the reference count on any keyrings it points to may be changed;
5. any keyrings it points to may be revoked, expired or have their security attributes changed; and
6. the contents of any keyrings to which it points may be changed (the whole point of keyrings being a shared set of credentials, modifiable by anyone with appropriate access).

To alter anything in the cred struct, the copy-and-replace principle must be adhered to. First take a copy, then alter the copy and then use RCU to change the task pointer to make it point to the new copy. There are wrappers to aid with this (see below).

A task may only alter its _own_ credentials; it is no longer permitted for a task to alter another's credentials. This means the `capset()` system call is no longer permitted to take any PID other than the one of the current process. Also `keyctl_instantiate()` and `keyctl_negate()` functions no longer permit attachment to process-specific keyrings in the requesting process as the instantiating process may need to create them.

1.4.1 Immutable Credentials

Once a set of credentials has been made public (by calling `commit_creds()` for example), it must be considered immutable, barring two exceptions:

1. The reference count may be altered.
2. While the keyring subscriptions of a set of credentials may not be changed, the keyrings subscribed to may have their contents altered.

To catch accidental credential alteration at compile time, struct task_struct has _const_ pointers to its credential sets, as does struct file. Furthermore, certain functions such as `get_cred()` and `put_cred()` operate on const pointers, thus rendering casts unnecessary, but require to temporarily ditch the const qualification to be able to alter the reference count.

1.4.2 Accessing Task Credentials

A task being able to alter only its own credentials permits the current process to read or replace its own credentials without the need for any form of locking -- which simplifies things greatly. It can just call:

```
const struct cred *current_cred()
```

to get a pointer to its credentials structure, and it doesn't have to release it afterwards.

There are convenience wrappers for retrieving specific aspects of a task's credentials (the value is simply returned in each case):

```
uid_t current_uid(void)         Current's real UID
gid_t current_gid(void)         Current's real GID
uid_t current_euid(void)        Current's effective UID
gid_t current_egid(void)        Current's effective GID
uid_t current_fsuid(void)       Current's file access UID
gid_t current_fsgid(void)       Current's file access GID
kernel_cap_t current_cap(void)  Current's effective capabilities
struct user_struct *current_user(void)  Current's user account
```

There are also convenience wrappers for retrieving specific associated pairs of a task's credentials:

```
void current_uid_gid(uid_t *, gid_t *);
void current_euid_egid(uid_t *, gid_t *);
void current_fsuid_fsgid(uid_t *, gid_t *);
```

which return these pairs of values through their arguments after retrieving them from the current task's credentials.

In addition, there is a function for obtaining a reference on the current process's current set of credentials:

```
const struct cred *get_current_cred(void);
```

and functions for getting references to one of the credentials that don't actually live in struct cred:

```
struct user_struct *get_current_user(void);
struct group_info *get_current_groups(void);
```

which get references to the current process's user accounting structure and supplementary groups list respectively.

Once a reference has been obtained, it must be released with `put_cred()`, `free_uid()` or `put_group_info()` as appropriate.

1.4.3 Accessing Another Task's Credentials

While a task may access its own credentials without the need for locking, the same is not true of a task wanting to access another task's credentials. It must use the RCU read lock and `rcu_dereference()`.

The `rcu_dereference()` is wrapped by:

```
const struct cred *__task_cred(struct task_struct *task);
```

This should be used inside the RCU read lock, as in the following example:

```
void foo(struct task_struct *t, struct foo_data *f)
{
        const struct cred *tcred;
        ...
        rcu_read_lock();
        tcred = __task_cred(t);
        f->uid = tcred->uid;
        f->gid = tcred->gid;
        f->groups = get_group_info(tcred->groups);
        rcu_read_unlock();
        ...
}
```

Should it be necessary to hold another task's credentials for a long period of time, and possibly to sleep while doing so, then the caller should get a reference on them using:

```
const struct cred *get_task_cred(struct task_struct *task);
```

This does all the RCU magic inside of it. The caller must call put_cred() on the credentials so obtained when they're finished with.

Note: The result of `__task_cred()` should not be passed directly to `get_cred()` as this may race with `commit_cred()`.

There are a couple of convenience functions to access bits of another task's credentials, hiding the RCU magic from the caller:

```
uid_t task_uid(task)            Task's real UID
uid_t task_euid(task)           Task's effective UID
```

If the caller is holding the RCU read lock at the time anyway, then:

```
__task_cred(task)->uid
__task_cred(task)->euid
```

should be used instead. Similarly, if multiple aspects of a task's credentials need to be accessed, RCU read lock should be used, `__task_cred()` called, the result stored in a temporary pointer and then the credential aspects called from that before dropping the lock. This prevents the potentially expensive RCU magic from being invoked multiple times.

Should some other single aspect of another task's credentials need to be accessed, then this can be used:

```
task_cred_xxx(task, member)
```

where 'member' is a non-pointer member of the cred struct. For instance:

```
uid_t task_cred_xxx(task, suid);
```

will retrieve 'struct cred::suid' from the task, doing the appropriate RCU magic. This may not be used for pointer members as what they point to may disappear the moment the RCU read lock is dropped.

1.4.4 Altering Credentials

As previously mentioned, a task may only alter its own credentials, and may not alter those of another task. This means that it doesn't need to use any locking to alter its own credentials.

To alter the current process's credentials, a function should first prepare a new set of credentials by calling:

```
struct cred *prepare_creds(void);
```

this locks current->cred_replace_mutex and then allocates and constructs a duplicate of the current process's credentials, returning with the mutex still held if successful. It returns NULL if not successful (out of memory).

The mutex prevents `ptrace()` from altering the ptrace state of a process while security checks on credentials construction and changing is taking place as the ptrace state may alter the outcome, particularly in the case of `execve()`.

The new credentials set should be altered appropriately, and any security checks and hooks done. Both the current and the proposed sets of credentials are available for this purpose as current_cred() will return the current set still at this point.

When replacing the group list, the new list must be sorted before it is added to the credential, as a binary search is used to test for membership. In practice, this means groups_sort() should be called before set_groups() or set_current_groups(). groups_sort() must not be called on a `struct group_list` which is shared as it may permute elements as part of the sorting process even if the array is already sorted.

When the credential set is ready, it should be committed to the current process by calling:

```
int commit_creds(struct cred *new);
```

This will alter various aspects of the credentials and the process, giving the LSM a chance to do likewise, then it will use `rcu_assign_pointer()` to actually commit the new credentials to `current->cred`, it will release `current->cred_replace_mutex` to allow `ptrace()` to take place, and it will notify the scheduler and others of the changes.

This function is guaranteed to return 0, so that it can be tail-called at the end of such functions as `sys_setresuid()`.

Note that this function consumes the caller's reference to the new credentials. The caller should _not_ call `put_cred()` on the new credentials afterwards.

Furthermore, once this function has been called on a new set of credentials, those credentials may _not_ be changed further.

Should the security checks fail or some other error occur after `prepare_creds()` has been called, then the following function should be invoked:

```
void abort_creds(struct cred *new);
```

This releases the lock on `current->cred_replace_mutex` that `prepare_creds()` got and then releases the new credentials.

A typical credentials alteration function would look something like this:

```
int alter_suid(uid_t suid)
{
        struct cred *new;
        int ret;

        new = prepare_creds();
        if (!new)
                return -ENOMEM;

        new->suid = suid;
        ret = security_alter_suid(new);
        if (ret < 0) {
                abort_creds(new);
                return ret;
        }

        return commit_creds(new);
}
```

1.4.5 Managing Credentials

There are some functions to help manage credentials:

- `void put_cred(const struct cred *cred);`

  This releases a reference to the given set of credentials. If the reference count reaches zero, the credentials will be scheduled for destruction by the RCU system.

- `const struct cred *get_cred(const struct cred *cred);`

  This gets a reference on a live set of credentials, returning a pointer to that set of credentials.

- `struct cred *get_new_cred(struct cred *cred);`

  This gets a reference on a set of credentials that is under construction and is thus still mutable, returning a pointer to that set of credentials.

1.5 Open File Credentials

When a new file is opened, a reference is obtained on the opening task's credentials and this is attached to the file struct as `f_cred` in place of `f_uid` and `f_gid`. Code that used to access `file->f_uid` and `file->f_gid` should now access `file->f_cred->fsuid` and `file->f_cred->fsgid`.

It is safe to access `f_cred` without the use of RCU or locking because the pointer will not change over the lifetime of the file struct, and nor will the contents of the cred struct pointed to, barring the exceptions listed above (see the Task Credentials section).

To avoid ”confused deputy” privilege escalation attacks, access control checks during subsequent operations on an opened file should use these credentials instead of ”current”'s credentials, as the file may have been passed to a more privileged process.

1.6 Overriding the VFS's Use of Credentials

Under some circumstances it is desirable to override the credentials used by the VFS, and that can be done by calling into such as `vfs_mkdir()` with a different set of credentials. This is done in the following places:

- `sys_faccessat()`.
- `do_coredump()`.
- nfs4recover.c.

CHAPTER

TWO

CONFIDENTIAL COMPUTING IN LINUX FOR X86 VIRTUALIZATION

By: Elena Reshetova <elena.reshetova@intel.com> and Carlos Bilbao <carlos.bilbao@amd.com>

2.1 Motivation

Kernel developers working on confidential computing for virtualized environments in x86 operate under a set of assumptions regarding the Linux kernel threat model that differ from the traditional view. Historically, the Linux threat model acknowledges attackers residing in userspace, as well as a limited set of external attackers that are able to interact with the kernel through various networking or limited HW-specific exposed interfaces (USB, thunderbolt). The goal of this document is to explain additional attack vectors that arise in the confidential computing space and discuss the proposed protection mechanisms for the Linux kernel.

2.2 Overview and terminology

Confidential Computing (CoCo) is a broad term covering a wide range of security technologies that aim to protect the confidentiality and integrity of data in use (vs. data at rest or data in transit). At its core, CoCo solutions provide a Trusted Execution Environment (TEE), where secure data processing can be performed and, as a result, they are typically further classified into different subtypes depending on the SW that is intended to be run in TEE. This document focuses on a subclass of CoCo technologies that are targeting virtualized environments and allow running Virtual Machines (VM) inside TEE. From now on in this document will be referring to this subclass of CoCo as 'Confidential Computing (CoCo) for the virtualized environments (VE)'.

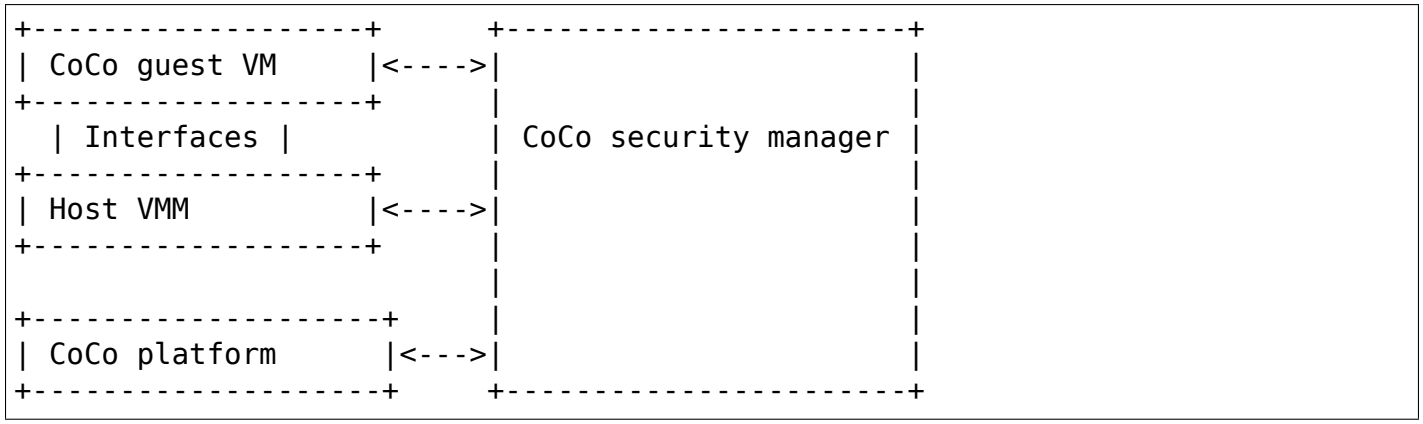
CoCo, in the virtualization context, refers to a set of HW and/or SW technologies that allow for stronger security guarantees for the SW running inside a CoCo VM. Namely, confidential computing allows its users to confirm the trustworthiness of all SW pieces to include in its

reduced Trusted Computing Base (TCB) given its ability to attest the state of these trusted components.

While the concrete implementation details differ between technologies, all available mechanisms aim to provide increased confidentiality and integrity for the VM's guest memory and execution state (vCPU registers), more tightly controlled guest interrupt injection, as well as some additional mechanisms to control guest-host page mapping. More details on the x86-specific solutions can be found in Intel Trust Domain Extensions (TDX) and AMD Memory Encryption.

The basic CoCo guest layout includes the host, guest, the interfaces that communicate guest and host, a platform capable of supporting CoCo VMs, and a trusted intermediary between the guest VM and the underlying platform that acts as a security manager. The host-side virtual machine monitor (VMM) typically consists of a subset of traditional VMM features and is still in charge of the guest lifecycle, i.e. create or destroy a CoCo VM, manage its access to system resources, etc. However, since it typically stays out of CoCo VM TCB, its access is limited to preserve the security objectives.

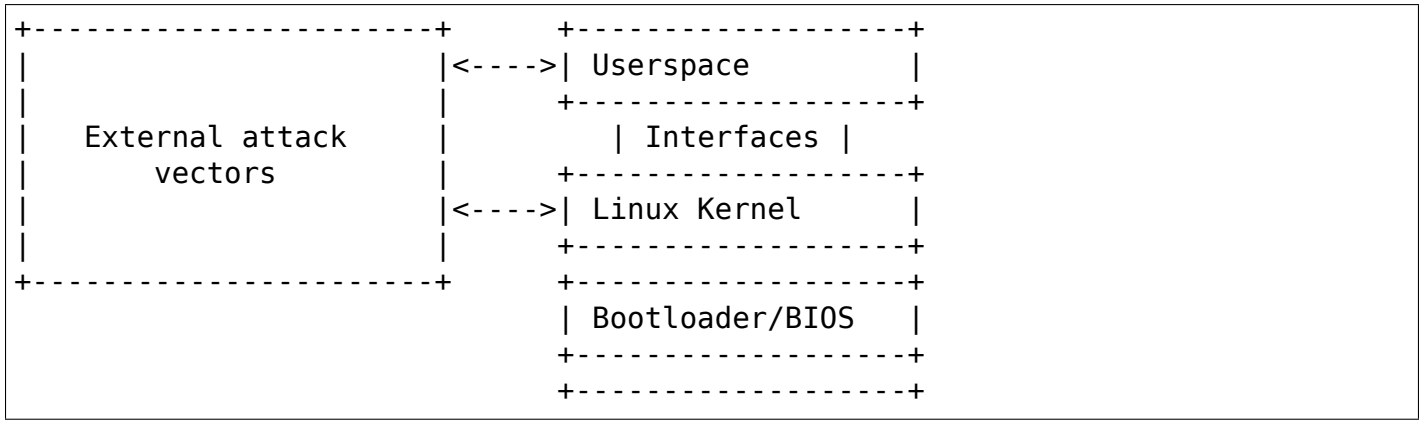
In the following diagram, the "<--->" lines represent bi-directional communication channels or interfaces between the CoCo security manager and the rest of the components (data flow for guest, host, hardware)

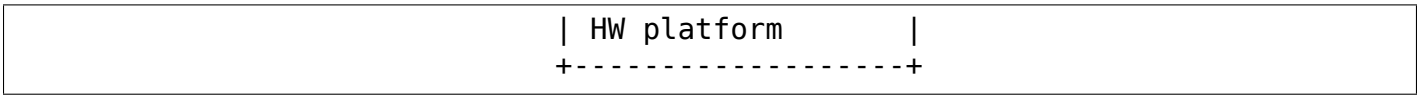


The specific details of the CoCo security manager vastly diverge between technologies. For example, in some cases, it will be implemented in HW while in others it may be pure SW.

2.3 Existing Linux kernel threat model

The overall components of the current Linux kernel threat model are:





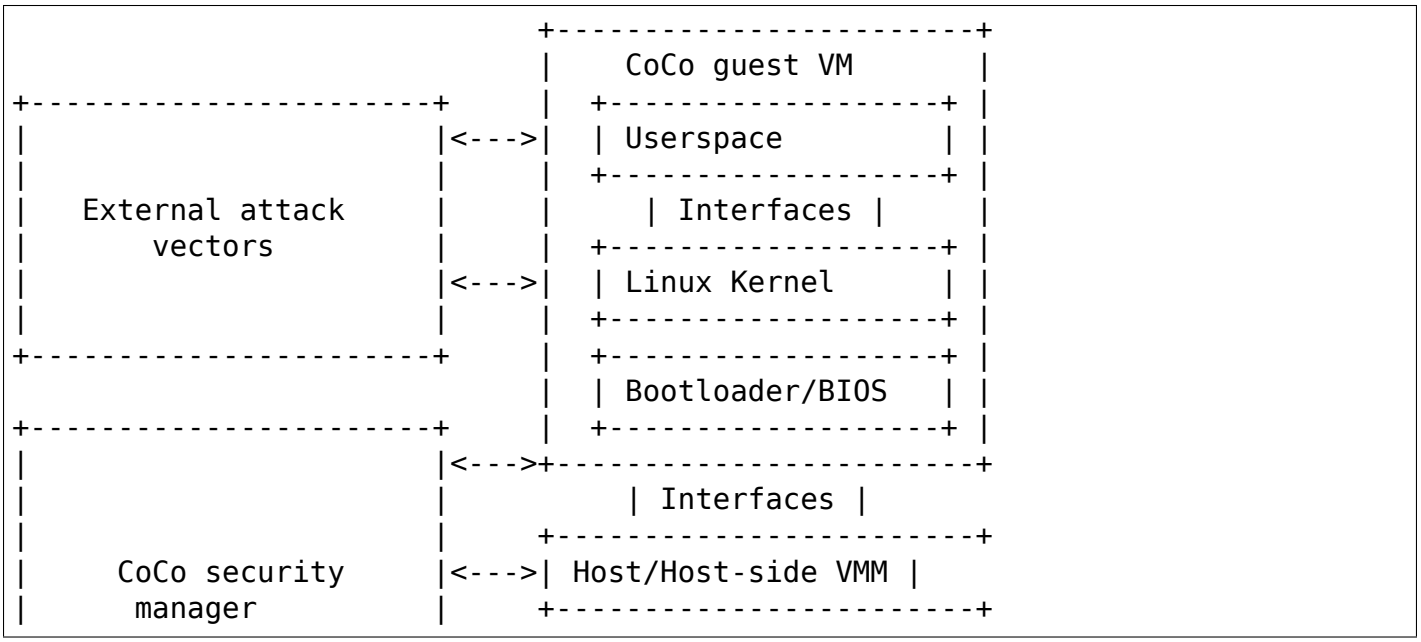
There is also communication between the bootloader and the kernel during the boot process, but this diagram does not represent it explicitly. The ”Interfaces” box represents the various interfaces that allow communication between kernel and userspace. This includes system calls, kernel APIs, device drivers, etc.

The existing Linux kernel threat model typically assumes execution on a trusted HW platform with all of the firmware and bootloaders included on its TCB. The primary attacker resides in the userspace, and all of the data coming from there is generally considered untrusted, unless userspace is privileged enough to perform trusted actions. In addition, external attackers are typically considered, including those with access to enabled external networks (e.g. Ethernet, Wireless, Bluetooth), exposed hardware interfaces (e.g. USB, Thunderbolt), and the ability to modify the contents of disks offline.

Regarding external attack vectors, it is interesting to note that in most cases external attackers will try to exploit vulnerabilities in userspace first, but that it is possible for an attacker to directly target the kernel; particularly if the host has physical access. Examples of direct kernel attacks include the vulnerabilities CVE-2019-19524, CVE-2022-0435 and CVE-2020-24490.

2.4 Confidential Computing threat model and its security objectives

Confidential Computing adds a new type of attacker to the above list: a potentially misbehaving host (which can also include some part of a traditional VMM or all of it), which is typically placed outside of the CoCo VM TCB due to its large SW attack surface. It is important to note that this doesn’t imply that the host or VMM are intentionally malicious, but that there exists a security value in having a small CoCo VM TCB. This new type of adversary may be viewed as a more powerful type of external attacker, as it resides locally on the same physical machine (in contrast to a remote network attacker) and has control over the guest kernel communication with most of the HW:



While traditionally the host has unlimited access to guest data and can leverage this access to attack the guest, the CoCo systems mitigate such attacks by adding security features like guest data confidentiality and integrity protection. This threat model assumes that those features are available and intact.

The **Linux kernel CoCo VM security objectives** can be summarized as follows:

1. Preserve the confidentiality and integrity of CoCo guest's private memory and registers.

2. Prevent privileged escalation from a host into a CoCo guest Linux kernel. While it is true that the host (and host-side VMM) requires some level of privilege to create, destroy, or pause the guest, part of the goal of preventing privileged escalation is to ensure that these operations do not provide a pathway for attackers to gain access to the guest's kernel.

The above security objectives result in two primary **Linux kernel CoCo VM assets**:

1. Guest kernel execution context.
2. Guest kernel private memory.

The host retains full control over the CoCo guest resources, and can deny access to them at any time. Examples of resources include CPU time, memory that the guest can consume, network bandwidth, etc. Because of this, the host Denial of Service (DoS) attacks against CoCo guests are beyond the scope of this threat model.

The **Linux CoCo VM attack surface** is any interface exposed from a CoCo guest Linux kernel towards an untrusted host that is not covered by the CoCo technology SW/HW protection. This includes any possible side-channels, as well as transient execution side channels. Examples of explicit (not side-channel) interfaces include accesses to port I/O, MMIO and DMA interfaces, access to PCI configuration space, VMM-specific hypercalls (towards Host-side VMM), access to shared memory pages, interrupts allowed to be injected into the guest kernel by the host, as well as CoCo technology-specific hypercalls, if present. Additionally, the host in a CoCo system typically controls the process of creating a CoCo guest: it has a method to load into a guest the firmware and bootloader images, the kernel image together with the kernel command line. All of this data should also be considered untrusted until its integrity and authenticity is established via attestation.

The table below shows a threat matrix for the CoCo guest Linux kernel but does not discuss potential mitigation strategies. The matrix refers to CoCo-specific versions of the guest, host and platform.

Table 1: CoCo Linux guest kernel threat matrix

Threat name	Threat description
Guest malicious configuration	A misbehaving host modifies one of the following guest's configuration: 1. Guest firmware or bootloader 2. Guest kernel or module binaries 3. Guest command line parameters This allows the host to break the integrity of the code running inside a CoCo guest, and violates the CoCo security objectives.
CoCo guest data attacks	A misbehaving host retains full control of the CoCo guest's data in-transit between the guest and the host-managed physical or virtual devices. This allows any attack against confidentiality, integrity or freshness of such data.
Malformed runtime input	A misbehaving host injects malformed input via any communication interface used by the guest's kernel code. If the code is not prepared to handle this input correctly, this can result in a host --> guest kernel privilege escalation. This includes traditional side-channel and/or transient execution attack vectors.
Malicious runtime input	A misbehaving host injects a specific input value via any communication interface used by the guest's kernel code. The difference with the previous attack vector (malformed runtime input) is that this input is not malformed, but its value is crafted to impact the guest's kernel security. Examples of such inputs include providing a malicious time to the guest or the entropy to the guest random number generator. Additionally, the timing of such events can be an attack vector on its own, if it results in a particular guest kernel action (i.e. processing of a host-injected interrupt). resistant to supplied host input.

CHAPTER

THREE

IMA TEMPLATE MANAGEMENT MECHANISM

3.1 Introduction

The original `ima` template is fixed length, containing the filedata hash and pathname. The filedata hash is limited to 20 bytes (md5/sha1). The pathname is a null terminated string, limited to 255 characters. To overcome these limitations and to add additional file metadata, it is necessary to extend the current version of IMA by defining additional templates. For example, information that could be possibly reported are the inode UID/GID or the LSM labels either of the inode and of the process that is accessing it.

However, the main problem to introduce this feature is that, each time a new template is defined, the functions that generate and display the measurements list would include the code for handling a new format and, thus, would significantly grow over the time.

The proposed solution solves this problem by separating the template management from the remaining IMA code. The core of this solution is the definition of two new data structures: a template descriptor, to determine which information should be included in the measurement list; a template field, to generate and display data of a given type.

Managing templates with these structures is very simple. To support a new data type, developers define the field identifier and implement two functions, init() and show(), respectively to generate and display measurement entries. Defining a new template descriptor requires specifying the template format (a string of field identifiers separated by the | character) through the `ima_template_fmt` kernel command line parameter. At boot time, IMA initializes the chosen template descriptor by translating the format into an array of template fields structures taken from the set of the supported ones.

After the initialization step, IMA will call `ima_alloc_init_template()` (new function defined within the patches for the new template management mechanism) to generate a new measurement entry by using the template descriptor chosen through the kernel configuration or through the newly introduced `ima_template` and `ima_template_fmt` kernel command line parameters. It is during this phase that the advantages of the new architecture are clearly shown: the latter function will not contain specific code to handle a given template but, instead, it simply calls the `init()` method of the template fields associated to the chosen template descriptor and store the result (pointer to allocated data and data length) in the measurement entry structure.

The same mechanism is employed to display measurements entries. The functions `ima[_ascii]_measurements_show()` retrieve, for each entry, the template descriptor used to produce that entry and call the show() method for each item of the array of template fields structures.

3.2 Supported Template Fields and Descriptors

In the following, there is the list of supported template fields (`'<identifier>': description`), that can be used to define new template descriptors by adding their identifier to the format string (support for more data types will be added later):

- 'd': the digest of the event (i.e. the digest of a measured file), calculated with the SHA1 or MD5 hash algorithm;
- 'n': the name of the event (i.e. the file name), with size up to 255 bytes;
- 'd-ng': the digest of the event, calculated with an arbitrary hash algorithm (field format: <hash algo>:digest);
- 'd-ngv2': same as d-ng, but prefixed with the ”ima” or ”verity” digest type (field format: <digest type>:<hash algo>:digest);
- 'd-modsig': the digest of the event without the appended modsig;
- 'n-ng': the name of the event, without size limitations;
- 'sig': the file signature, based on either the file's/fsverity's digest[1], or the EVM portable signature, if 'security.ima' contains a file hash.
- 'modsig' the appended file signature;
- 'buf': the buffer data that was used to generate the hash without size limitations;
- 'evmsig': the EVM portable signature;
- 'iuid': the inode UID;
- 'igid': the inode GID;
- 'imode': the inode mode;
- **'xattrnames': a list of xattr names (separated by |), only if the xattr is** present;
- 'xattrlengths': a list of xattr lengths (u32), only if the xattr is present;
- 'xattrvalues': a list of xattr values;

Below, there is the list of defined template descriptors:

- ”ima”: its format is `d|n`;
- ”ima-ng” (default): its format is `d-ng|n-ng`;
- ”ima-ngv2”: its format is `d-ngv2|n-ng`;
- ”ima-sig”: its format is `d-ng|n-ng|sig`;
- ”ima-sigv2”: its format is `d-ngv2|n-ng|sig`;
- ”ima-buf”: its format is `d-ng|n-ng|buf`;
- ”ima-modsig”: its format is `d-ng|n-ng|sig|d-modsig|modsig`;
- ”evm-sig”: its format is `d-ng|n-ng|evmsig|xattrnames|xattrlengths|xattrvalues|iuid|igic`

3.3 Use

To specify the template descriptor to be used to generate measurement entries, currently the following methods are supported:

- select a template descriptor among those supported in the kernel configuration (`ima-ng` is the default choice);
- specify a template descriptor name from the kernel command line through the `ima_template=` parameter;
- register a new template descriptor with custom format through the kernel command line parameter `ima_template_fmt=`.

CHAPTER

FOUR

KERNEL KEYS

4.1 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:

4.1.1 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”.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 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.

4.1.6 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 *Key Request Service*.

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 implementation 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/core-api/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)

4.1.7 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 *Key Request Service*.

- 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.

4.1.8 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);
```

4.1.9 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.

4.1.10 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.

4.1.11 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
```

4.2 Encrypted keys for the eCryptfs filesystem

ECryptfs is a stacked filesystem which transparently encrypts and decrypts each file using a randomly generated File Encryption Key (FEK).

Each FEK is in turn encrypted with a File Encryption Key Encryption Key (FEKEK) either in kernel space or in user space with a daemon called 'ecryptfsd'. In the former case the operation is performed directly by the kernel CryptoAPI using a key, the FEKEK, derived from a user prompted passphrase; in the latter the FEK is encrypted by 'ecryptfsd' with the help of external libraries in order to support other mechanisms like public key cryptography, PKCS#11 and TPM based operations.

The data structure defined by eCryptfs to contain information required for the FEK decryption is called authentication token and, currently, can be stored in a kernel key of the 'user' type, inserted in the user's session specific keyring by the userspace utility 'mount.ecryptfs' shipped with the package 'ecryptfs-utils'.

The 'encrypted' key type has been extended with the introduction of the new format 'ecryptfs' in order to be used in conjunction with the eCryptfs filesystem. Encrypted keys of the newly introduced format store an authentication token in its payload with a FEKEK randomly generated by the kernel and protected by the parent master key.

In order to avoid known-plaintext attacks, the datablob obtained through commands 'keyctl print' or 'keyctl pipe' does not contain the overall authentication token, which content is well known, but only the FEKEK in encrypted form.

The eCryptfs filesystem may really benefit from using encrypted keys in that the required key can be securely generated by an Administrator and provided at boot time after the unsealing of a 'trusted' key in order to perform the mount in a controlled environment. Another advantage is that the key is not exposed to threats of malicious software, because it is available in clear form only at kernel level.

Usage:

```
keyctl add encrypted name "new ecryptfs key-type:master-key-name keylen" ring
keyctl add encrypted name "load hex_blob" ring
keyctl update keyid "update key-type:master-key-name"
```

Where:

```
name:= '<16 hexadecimal characters>'
key-type:= 'trusted' | 'user'
keylen:= 64
```

Example of encrypted key usage with the eCryptfs filesystem:

Create an encrypted key ”1000100010001000” of length 64 bytes with format 'ecryptfs' and save it using a previously loaded user key ”test”:

```
$ keyctl add encrypted 1000100010001000 "new ecryptfs user:test 64" @u
19184530

$ keyctl print 19184530
ecryptfs user:test 64 490045d4bfe48c99f0d465fbbbb79e7500da954178e2de0697
dd85091f5450a0511219e9f7cd70dcd498038181466f78ac8d4c19504fcc72402bfc41c2
f253a41b7507ccaa4b2b03fff19a69d1cc0b16e71746473f023a95488b6edfd86f7fdd40
9d292e4bacded1258880122dd553a661

$ keyctl pipe 19184530 > ecryptfs.blob
```

Mount an eCryptfs filesystem using the created encrypted key ”1000100010001000” into the '/secret' directory:

```
$ mount -i -t ecryptfs -oecryptfs_sig=1000100010001000,\
  ecryptfs_cipher=aes,ecryptfs_key_bytes=32 /secret /secret
```

4.3 Key Request Service

The key request service is part of the key retention service (refer to *Kernel Key Retention Service*). This document explains more fully how the requesting algorithm works.

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

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

or:

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

or:

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

or:

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

Or by userspace invoking the request_key system call:

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

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

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

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

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

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

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

4.3.1 The Process

A request proceeds in the following manner:

1) Process A calls request_key() [the userspace syscall calls the kernel interface].

2) request_key() searches the process's subscribed keyrings to see if there's a suitable key there. If there is, it returns the key. If there isn't, and callout_info is not set, an error is returned. Otherwise the process proceeds to the next step.

3) request_key() sees that A doesn't have the desired key yet, so it creates two things:

    a) An uninstantiated key U of requested type and description.

   b) An authorisation key V that refers to key U and notes that process A is the context in which key U should be instantiated and secured, and from which associated key requests may be satisfied.

4) request_key() then forks and executes /sbin/request-key with a new session keyring that contains a link to auth key V.

5) /sbin/request-key assumes the authority associated with key U.

6) /sbin/request-key execs an appropriate program to perform the actual instantiation.

7) The program may want to access another key from A's context (say a Kerberos TGT key). It just requests the appropriate key, and the keyring search notes that the session keyring has auth key V in its bottom level.

   This will permit it to then search the keyrings of process A with the UID, GID, groups and security info of process A as if it was process A, and come up with key W.

8) The program then does what it must to get the data with which to instantiate key U, using key W as a reference (perhaps it contacts a Kerberos server using the TGT) and then instantiates key U.

9) Upon instantiating key U, auth key V is automatically revoked so that it may not be used again.

10) The program then exits 0 and request_key() deletes key V and returns key U to the caller.

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

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

4.3.2 Negative Instantiation And Rejection

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

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

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

4.3.3 The Search Algorithm

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

1) When the key management code searches for a key (keyring_search_rcu) it firstly calls key_permission(SEARCH) on the keyring it's starting with, if this denies permission, it doesn't search further.

2) It considers all the non-keyring keys within that keyring and, if any key matches the criteria specified, calls key_permission(SEARCH) on it to see if the key is allowed to be found. If it is, that key is returned; if not, the search continues, and the error code is retained if of higher priority than the one currently set.

3) It then considers all the keyring-type keys in the keyring it's currently searching. It calls key_permission(SEARCH) on each keyring, and if this grants permission, it recurses, executing steps (2) and (3) on that keyring.

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

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

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

1) If extant, the process's thread keyring is searched.

2) If extant, the process's process keyring is searched.

3) The process's session keyring is searched.

4) If the process has assumed the authority associated with a request_key() authorisation key then:

   a) If extant, the calling process's thread keyring is searched.

   b) If extant, the calling process's process keyring is searched.

   c) The calling process's session keyring is searched.

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

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

The error priority is:

```
EKEYREVOKED > EKEYEXPIRED > ENOKEY
```

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

4.4 Trusted and Encrypted Keys

Trusted and Encrypted Keys are two new key types added to the existing kernel key ring service. Both of these new types are variable length symmetric keys, and in both cases all keys are created in the kernel, and user space sees, stores, and loads only encrypted blobs. Trusted Keys require the availability of a Trust Source for greater security, while Encrypted Keys can be used on any system. All user level blobs, are displayed and loaded in hex ASCII for convenience, and are integrity verified.

4.4.1 Trust Source

A trust source provides the source of security for Trusted Keys. This section lists currently supported trust sources, along with their security considerations. Whether or not a trust source is sufficiently safe depends on the strength and correctness of its implementation, as well as the threat environment for a specific use case. Since the kernel doesn't know what the environment is, and there is no metric of trust, it is dependent on the consumer of the Trusted Keys to determine if the trust source is sufficiently safe.

- Root of trust for storage

  (1) TPM (Trusted Platform Module: hardware device)

  Rooted to Storage Root Key (SRK) which never leaves the TPM that provides crypto operation to establish root of trust for storage.

  (2) TEE (Trusted Execution Environment: OP-TEE based on Arm TrustZone)

  Rooted to Hardware Unique Key (HUK) which is generally burnt in on-chip fuses and is accessible to TEE only.

  (3) CAAM (Cryptographic Acceleration and Assurance Module: IP on NXP SoCs)

  When High Assurance Boot (HAB) is enabled and the CAAM is in secure mode, trust is rooted to the OTPMK, a never-disclosed 256-bit key randomly generated and fused into each SoC at manufacturing time. Otherwise, a common fixed test key is used instead.

- Execution isolation

  (1) TPM

  Fixed set of operations running in isolated execution environment.

  (2) TEE

  Customizable set of operations running in isolated execution environment verified via Secure/Trusted boot process.

  (3) CAAM

  Fixed set of operations running in isolated execution environment.

- Optional binding to platform integrity state

  (1) TPM

  Keys can be optionally sealed to specified PCR (integrity measurement) values, and only unsealed by the TPM, if PCRs and blob integrity verifications match. A loaded Trusted Key can be updated with new (future) PCR values, so keys are easily migrated

to new PCR values, such as when the kernel and initramfs are updated. The same key can have many saved blobs under different PCR values, so multiple boots are easily supported.

(2) TEE

Relies on Secure/Trusted boot process for platform integrity. It can be extended with TEE based measured boot process.

(3) CAAM

Relies on the High Assurance Boot (HAB) mechanism of NXP SoCs for platform integrity.

- Interfaces and APIs

(1) TPM

TPMs have well-documented, standardized interfaces and APIs.

(2) TEE

TEEs have well-documented, standardized client interface and APIs. For more details refer to `Documentation/driver-api/tee.rst`.

(3) CAAM

Interface is specific to silicon vendor.

- Threat model

The strength and appropriateness of a particular trust source for a given purpose must be assessed when using them to protect security-relevant data.

4.4.2 Key Generation

Trusted Keys

New keys are created from random numbers. They are encrypted/decrypted using a child key in the storage key hierarchy. Encryption and decryption of the child key must be protected by a strong access control policy within the trust source. The random number generator in use differs according to the selected trust source:

- TPM: hardware device based RNG

Keys are generated within the TPM. Strength of random numbers may vary from one device manufacturer to another.

- TEE: OP-TEE based on Arm TrustZone based RNG

RNG is customizable as per platform needs. It can either be direct output from platform specific hardware RNG or a software based Fortuna CSPRNG which can be seeded via multiple entropy sources.

- CAAM: Kernel RNG

The normal kernel random number generator is used. To seed it from the CAAM HWRNG, enable CRYPTO_DEV_FSL_CAAM_RNG_API and ensure the device is probed.

Users may override this by specifying `trusted.rng=kernel` on the kernel command-line to override the used RNG with the kernel's random number pool.

Encrypted Keys

Encrypted keys do not depend on a trust source, and are faster, as they use AES for encryption/decryption. New keys are created either from kernel-generated random numbers or user-provided decrypted data, and are encrypted/decrypted using a specified 'master' key. The 'master' key can either be a trusted-key or user-key type. The main disadvantage of encrypted keys is that if they are not rooted in a trusted key, they are only as secure as the user key encrypting them. The master user key should therefore be loaded in as secure a way as possible, preferably early in boot.

4.4.3 Usage

Trusted Keys usage: TPM

TPM 1.2: By default, trusted keys are sealed under the SRK, which has the default authorization value (20 bytes of 0s). This can be set at takeownership time with the TrouSerS utility: "tpm_takeownership -u -z".

TPM 2.0: The user must first create a storage key and make it persistent, so the key is available after reboot. This can be done using the following commands.

With the IBM TSS 2 stack:

```
#> tsscreateprimary -hi o -st
Handle 80000000
#> tssevictcontrol -hi o -ho 80000000 -hp 81000001
```

Or with the Intel TSS 2 stack:

```
#> tpm2_createprimary --hierarchy o -G rsa2048 -c key.ctxt
[...]
#> tpm2_evictcontrol -c key.ctxt 0x81000001
persistentHandle: 0x81000001
```

Usage:

```
keyctl add trusted name "new keylen [options]" ring
keyctl add trusted name "load hex_blob [pcrlock=pcrnum]" ring
keyctl update key "update [options]"
keyctl print keyid

options:
   keyhandle=    ascii hex value of sealing key
                   TPM 1.2: default 0x40000000 (SRK)
                   TPM 2.0: no default; must be passed every time
   keyauth=      ascii hex auth for sealing key default 0x00...i
                 (40 ascii zeros)
   blobauth=     ascii hex auth for sealed data default 0x00...
```

```
                (40 ascii zeros)
  pcrinfo=      ascii hex of PCR_INFO or PCR_INFO_LONG (no default)
  pcrlock=      pcr number to be extended to "lock" blob
  migratable=   0|1 indicating permission to reseal to new PCR values,
                default 1 (resealing allowed)
  hash=         hash algorithm name as a string. For TPM 1.x the only
                allowed value is sha1. For TPM 2.x the allowed values
                are sha1, sha256, sha384, sha512 and sm3-256.
  policydigest= digest for the authorization policy. must be calculated
                with the same hash algorithm as specified by the 'hash='
                option.
  policyhandle= handle to an authorization policy session that defines the
                same policy and with the same hash algorithm as was used to
                seal the key.
```

"keyctl print" returns an ascii hex copy of the sealed key, which is in standard TPM_STORED_DATA format. The key length for new keys are always in bytes. Trusted Keys can be 32 - 128 bytes (256 - 1024 bits), the upper limit is to fit within the 2048 bit SRK (RSA) keylength, with all necessary structure/padding.

Trusted Keys usage: TEE

Usage:

```
keyctl add trusted name "new keylen" ring
keyctl add trusted name "load hex_blob" ring
keyctl print keyid
```

"keyctl print" returns an ASCII hex copy of the sealed key, which is in format specific to TEE device implementation. The key length for new keys is always in bytes. Trusted Keys can be 32 - 128 bytes (256 - 1024 bits).

Trusted Keys usage: CAAM

Usage:

```
keyctl add trusted name "new keylen" ring
keyctl add trusted name "load hex_blob" ring
keyctl print keyid
```

"keyctl print" returns an ASCII hex copy of the sealed key, which is in a CAAM-specific format. The key length for new keys is always in bytes. Trusted Keys can be 32 - 128 bytes (256 - 1024 bits).

Encrypted Keys usage

The decrypted portion of encrypted keys can contain either a simple symmetric key or a more complex structure. The format of the more complex structure is application specific, which is identified by 'format'.

Usage:

```
keyctl add encrypted name "new [format] key-type:master-key-name keylen"
    ring
keyctl add encrypted name "new [format] key-type:master-key-name keylen
    decrypted-data" ring
keyctl add encrypted name "load hex_blob" ring
keyctl update keyid "update key-type:master-key-name"
```

Where:

```
format:= 'default | ecryptfs | enc32'
key-type:= 'trusted' | 'user'
```

Examples of trusted and encrypted key usage

Create and save a trusted key named ”kmk” of length 32 bytes.

Note: When using a TPM 2.0 with a persistent key with handle 0x81000001, append 'keyhandle=0x81000001' to statements between quotes, such as ”new 32 keyhandle=0x81000001”.

```
$ keyctl add trusted kmk "new 32" @u
440502848

$ keyctl show
Session Keyring
       -3 --alswrv    500   500  keyring: _ses
 97833714 --alswrv    500    -1   \_ keyring: _uid.500
440502848 --alswrv    500   500       \_ trusted: kmk

$ keyctl print 440502848
0101000000000000000001005d01b7e3f4a6be5709930f3b70a743cbb42e0cc95e18e915
3f60da455bbf1144ad12e4f92b452f966929f6105fd29ca28e4d4d5a031d068478bacb0b
27351119f822911b0a11ba3d3498ba6a32e50dac7f32894dd890eb9ad578e4e292c83722
a52e56a097e6a68b3f56f7a52ece0cdccba1eb62cad7d817f6dc58898b3ac15f36026fec
d568bd4a706cb60bb37be6d8f1240661199d640b66fb0fe3b079f97f450b9ef9c22c6d5d
dd379f0facd1cd020281dfa3c70ba21a3fa6fc2471dc6d13ecf8298b946f65345faa5ef0
f1f8fff03ad0acb083725535636addb08d73dedb9832da198081e5deae84bfaf0409c22b
e4a8aea2b607ec96931e6f4d4fe563ba

$ keyctl pipe 440502848 > kmk.blob
```

Load a trusted key from the saved blob:

```
$ keyctl add trusted kmk "load `cat kmk.blob`" @u
268728824
```

```
$ keyctl print 268728824
0101000000000000000001005d01b7e3f4a6be5709930f3b70a743cbb42e0cc95e18e915
3f60da455bbf1144ad12e4f92b452f966929f6105fd29ca28e4d4d5a031d068478bacb0b
27351119f822911b0a11ba3d3498ba6a32e50dac7f32894dd890eb9ad578e4e292c83722
a52e56a097e6a68b3f56f7a52ece0cdccba1eb62cad7d817f6dc58898b3ac15f36026fec
d568bd4a706cb60bb37be6d8f1240661199d640b66fb0fe3b079f97f450b9ef9c22c6d5d
dd379f0facd1cd020281dfa3c70ba21a3fa6fc2471dc6d13ecf8298b946f65345faa5ef0
f1f8fff03ad0acb083725535636addb08d73dedb9832da198081e5deae84bfaf0409c22b
e4a8aea2b607ec96931e6f4d4fe563ba
```

Reseal (TPM specific) a trusted key under new PCR values:

```
$ keyctl update 268728824 "update pcrinfo=`cat pcr.blob`"
$ keyctl print 268728824
010100000000002c0002800093c35a09b70fff26e7a98ae786c641e678ec6ffb6b46d805
77c8a6377aed9d3219c6dfec4b23ffe3000001005d37d472ac8a44023fbb3d18583a4f73
d3a076c0858f6f1dcaa39ea0f119911ff03f5406df4f7f27f41da8d7194f45c9f4e00f2e
df449f266253aa3f52e55c53de147773e00f0f9aca86c64d94c95382265968c354c5eab4
9638c5ae99c89de1e0997242edfb0b501744e11ff9762dfd951cffd93227cc513384e7e6
e782c29435c7ec2edafaa2f4c1fe6e7a781b59549ff5296371b42133777dcc5b8b971610
94bc67ede19e43ddb9dc2baacad374a36feaf0314d700af0a65c164b7082401740e489c9
7ef6a24defe4846104209bf0c3eced7fa1a672ed5b125fc9d8cd88b476a658a4434644ef
df8ae9a178e9f83ba9f08d10fa47e4226b98b0702f06b3b8
```

The initial consumer of trusted keys is EVM, which at boot time needs a high quality symmetric key for HMAC protection of file metadata. The use of a trusted key provides strong guarantees that the EVM key has not been compromised by a user level problem, and when sealed to a platform integrity state, protects against boot and offline attacks. Create and save an encrypted key ”evm” using the above trusted key ”kmk”:

option 1: omitting 'format':

```
$ keyctl add encrypted evm "new trusted:kmk 32" @u
159771175
```

option 2: explicitly defining 'format' as 'default':

```
$ keyctl add encrypted evm "new default trusted:kmk 32" @u
159771175

$ keyctl print 159771175
default trusted:kmk 32 2375725ad57798846a9bbd240de8906f006e66c03af53b1b3
82dbbc55be2a44616e4959430436dc4f2a7a9659aa60bb4652aeb2120f149ed197c564e0
24717c64 5972dcb82ab2dde83376d82b2e3c09ffc

$ keyctl pipe 159771175 > evm.blob
```

Load an encrypted key ”evm” from saved blob:

```
$ keyctl add encrypted evm "load `cat evm.blob`" @u
831684262
```

```
$ keyctl print 831684262
default trusted:kmk 32 2375725ad57798846a9bbd240de8906f006e66c03af53b1b3
82dbbc55be2a44616e4959430436dc4f2a7a9659aa60bb4652aeb2120f149ed197c564e0
24717c64 5972dcb82ab2dde83376d82b2e3c09ffc
```

Instantiate an encrypted key "evm" using user-provided decrypted data:

```
$ evmkey=$(dd if=/dev/urandom bs=1 count=32 | xxd -c32 -p)
$ keyctl add encrypted evm "new default user:kmk 32 $evmkey" @u
794890253

$ keyctl print 794890253
default user:kmk 32 2375725ad57798846a9bbd240de8906f006e66c03af53b1b382d
bbc55be2a44616e4959430436dc4f2a7a9659aa60bb4652aeb2120f149ed197c564e0247
17c64 5972dcb82ab2dde83376d82b2e3c09ffc
```

Other uses for trusted and encrypted keys, such as for disk and file encryption are anticipated. In particular the new format 'ecryptfs' has been defined in order to use encrypted keys to mount an eCryptfs filesystem. More details about the usage can be found in the file `Documentation/security/keys/ecryptfs.rst`.

Another new format 'enc32' has been defined in order to support encrypted keys with payload size of 32 bytes. This will initially be used for nvdimm security but may expand to other usages that require 32 bytes payload.

TPM 2.0 ASN.1 Key Format

The TPM 2.0 ASN.1 key format is designed to be easily recognisable, even in binary form (fixing a problem we had with the TPM 1.2 ASN.1 format) and to be extensible for additions like importable keys and policy:

```
TPMKey ::= SEQUENCE {
    type            OBJECT IDENTIFIER
    emptyAuth       [0] EXPLICIT BOOLEAN OPTIONAL
    parent          INTEGER
    pubkey          OCTET STRING
    privkey         OCTET STRING
}
```

type is what distinguishes the key even in binary form since the OID is provided by the TCG to be unique and thus forms a recognizable binary pattern at offset 3 in the key. The OIDs currently made available are:

```
2.23.133.10.1.3 TPM Loadable key.  This is an asymmetric key (Usually
                RSA2048 or Elliptic Curve) which can be imported by a
                TPM2_Load() operation.

2.23.133.10.1.4 TPM Importable Key.  This is an asymmetric key (Usually
                RSA2048 or Elliptic Curve) which can be imported by a
                TPM2_Import() operation.
```

```
2.23.133.10.1.5 TPM Sealed Data.  This is a set of data (up to 128
                bytes) which is sealed by the TPM.  It usually
                represents a symmetric key and must be unsealed before
                use.
```

The trusted key code only uses the TPM Sealed Data OID.

emptyAuth is true if the key has well known authorization "". If it is false or not present, the key requires an explicit authorization phrase. This is used by most user space consumers to decide whether to prompt for a password.

parent represents the parent key handle, either in the 0x81 MSO space, like 0x81000001 for the RSA primary storage key. Userspace programmes also support specifying the primary handle in the 0x40 MSO space. If this happens the Elliptic Curve variant of the primary key using the TCG defined template will be generated on the fly into a volatile object and used as the parent. The current kernel code only supports the 0x81 MSO form.

pubkey is the binary representation of TPM2B_PRIVATE excluding the initial TPM2B header, which can be reconstructed from the ASN.1 octet string length.

privkey is the binary representation of TPM2B_PUBLIC excluding the initial TPM2B header which can be reconstructed from the ASN.1 octed string length.

CHAPTER

FIVE

LINUX SECURITY MODULES: GENERAL SECURITY HOOKS FOR LINUX

Author
Stephen Smalley

Author
Timothy Fraser

Author
Chris Vance

Note: The APIs described in this book are outdated.

5.1 Introduction

In March 2001, the National Security Agency (NSA) gave a presentation about Security-Enhanced Linux (SELinux) at the 2.5 Linux Kernel Summit. SELinux is an implementation of flexible and fine-grained nondiscretionary access controls in the Linux kernel, originally implemented as its own particular kernel patch. Several other security projects (e.g. RSBAC, Medusa) have also developed flexible access control architectures for the Linux kernel, and various projects have developed particular access control models for Linux (e.g. LIDS, DTE, SubDomain). Each project has developed and maintained its own kernel patch to support its security needs.

In response to the NSA presentation, Linus Torvalds made a set of remarks that described a security framework he would be willing to consider for inclusion in the mainstream Linux kernel. He described a general framework that would provide a set of security hooks to control operations on kernel objects and a set of opaque security fields in kernel data structures for maintaining security attributes. This framework could then be used by loadable kernel modules to implement any desired model of security. Linus also suggested the possibility of migrating the Linux capabilities code into such a module.

The Linux Security Modules (LSM) project was started by WireX to develop such a framework. LSM was a joint development effort by several security projects, including Immunix, SELinux, SGI and Janus, and several individuals, including Greg Kroah-Hartman and James Morris, to develop a Linux kernel patch that implements this framework. The work was incorporated in the mainstream in December of 2003. This technical report provides an overview of the framework and the capabilities security module.

5.2 LSM Framework

The LSM framework provides a general kernel framework to support security modules. In particular, the LSM framework is primarily focused on supporting access control modules, although future development is likely to address other security needs such as sandboxing. By itself, the framework does not provide any additional security; it merely provides the infrastructure to support security modules. The LSM framework is optional, requiring *CONFIG_SECURITY* to be enabled. The capabilities logic is implemented as a security module. This capabilities module is discussed further in *LSM Capabilities Module*.

The LSM framework includes security fields in kernel data structures and calls to hook functions at critical points in the kernel code to manage the security fields and to perform access control. It also adds functions for registering security modules. An interface */sys/kernel/security/lsm* reports a comma separated list of security modules that are active on the system.

The LSM security fields are simply `void*` pointers. The data is referred to as a blob, which may be managed by the framework or by the individual security modules that use it. Security blobs that are used by more than one security module are typically managed by the framework. For process and program execution security information, security fields are included in `struct task_struct` and `struct cred`. For filesystem security information, a security field is included in `struct super_block`. For pipe, file, and socket security information, security fields are included in `struct inode` and `struct file`. For System V IPC security information, security fields were added to `struct kern_ipc_perm` and `struct msg_msg`; additionally, the definitions for `struct msg_msg`, struct msg_queue, and struct shmid_kernel were moved to header files (`include/linux/msg.h` and `include/linux/shm.h` as appropriate) to allow the security modules to use these definitions.

For packet and network device security information, security fields were added to `struct sk_buff` and `struct scm_cookie`. Unlike the other security module data, the data used here is a 32-bit integer. The security modules are required to map or otherwise associate these values with real security attributes.

LSM hooks are maintained in lists. A list is maintained for each hook, and the hooks are called in the order specified by CONFIG_LSM. Detailed documentation for each hook is included in the *security/security.c* source file.

The LSM framework provides for a close approximation of general security module stacking. It defines security_add_hooks() to which each security module passes a `struct security_hooks_list`, which are added to the lists. The LSM framework does not provide a mechanism for removing hooks that have been registered. The SELinux security module has implemented a way to remove itself, however the feature has been deprecated.

The hooks can be viewed as falling into two major categories: hooks that are used to manage the security fields and hooks that are used to perform access control. Examples of the first category of hooks include the security_inode_alloc() and security_inode_free() These hooks are used to allocate and free security structures for inode objects. An example of the second category of hooks is the security_inode_permission() hook. This hook checks permission when accessing an inode.

5.3 LSM Capabilities Module

The POSIX.1e capabilities logic is maintained as a security module stored in the file `security/commoncap.c`. The capabilities module uses the order field of the `lsm_info` description to identify it as the first security module to be registered. The capabilities security module does not use the general security blobs, unlike other modules. The reasons are historical and are based on overhead, complexity and performance concerns.

CHAPTER

SIX

LINUX SECURITY MODULE DEVELOPMENT

Based on https://lore.kernel.org/r/20071026073721.618b4778@laptopd505.fenrus.org, a new LSM is accepted into the kernel when its intent (a description of what it tries to protect against and in what cases one would expect to use it) has been appropriately documented in `Documentation/admin-guide/LSM/`. This allows an LSM's code to be easily compared to its goals, and so that end users and distros can make a more informed decision about which LSMs suit their requirements.

For extensive documentation on the available LSM hook interfaces, please see `security/security.c` and associated structures:

void **security_free_mnt_opts**(void **mnt_opts)

Free memory associated with mount options

Parameters

void **mnt_opts

LSM processed mount options

Description

Free memory associated with **mnt_ops**.

int **security_sb_eat_lsm_opts**(char *options, void **mnt_opts)

Consume LSM mount options

Parameters

char *options

mount options

void **mnt_opts

LSM processed mount options

Description

Eat (scan **options**) and save them in **mnt_opts**.

Return

Returns 0 on success, negative values on failure.

int **security_sb_mnt_opts_compat**(struct super_block *sb, void *mnt_opts)

Check if new mount options are allowed

Parameters

struct super_block *sb
filesystem superblock

void *mnt_opts
new mount options

Description

Determine if the new mount options in **mnt_opts** are allowed given the existing mounted filesystem at **sb**. **sb** superblock being compared.

Return

Returns 0 if options are compatible.

int **security_sb_remount**(struct super_block *sb, void *mnt_opts)
Verify no incompatible mount changes during remount

Parameters

struct super_block *sb
filesystem superblock

void *mnt_opts
(re)mount options

Description

Extracts security system specific mount options and verifies no changes are being made to those options.

Return

Returns 0 if permission is granted.

int **security_sb_set_mnt_opts**(struct super_block *sb, void *mnt_opts, unsigned long kern_flags, unsigned long *set_kern_flags)
Set the mount options for a filesystem

Parameters

struct super_block *sb
filesystem superblock

void *mnt_opts
binary mount options

unsigned long kern_flags
kernel flags (in)

unsigned long *set_kern_flags
kernel flags (out)

Description

Set the security relevant mount options used for a superblock.

Return

Returns 0 on success, error on failure.

int **security_sb_clone_mnt_opts**(const struct super_block *oldsb, struct super_block *newsb, unsigned long kern_flags, unsigned long *set_kern_flags)

Duplicate superblock mount options

Parameters

const struct super_block *oldsb
source superblock

struct super_block *newsb
destination superblock

unsigned long kern_flags
kernel flags (in)

unsigned long *set_kern_flags
kernel flags (out)

Description

Copy all security options from a given superblock to another.

Return

Returns 0 on success, error on failure.

int **security_dentry_init_security**(struct *dentry* *dentry, int mode, const struct qstr *name, const char **xattr_name, void **ctx, u32 *ctxlen)

Perform dentry initialization

Parameters

struct dentry *dentry
the dentry to initialize

int mode
mode used to determine resource type

const struct qstr *name
name of the last path component

const char **xattr_name
name of the security/LSM xattr

void **ctx
pointer to the resulting LSM context

u32 *ctxlen
length of **ctx**

Description

Compute a context for a dentry as the inode is not yet available since NFSv4 has no label backed by an EA anyway. It is important to note that **xattr_name** does not need to be free'd by the caller, it is a static string.

Return

Returns 0 on success, negative values on failure.

int **security_dentry_create_files_as**(struct *dentry* *dentry, int mode, struct qstr *name, const struct cred *old, struct cred *new)

Perform dentry initialization

Parameters

struct dentry *dentry
the dentry to initialize

int mode
mode used to determine resource type

struct qstr *name
name of the last path component

const struct cred *old
creds to use for LSM context calculations

struct cred *new
creds to modify

Description

Compute a context for a dentry as the inode is not yet available and set that context in passed in creds so that new files are created using that context. Context is calculated using the passed in creds and not the creds of the caller.

Return

Returns 0 on success, error on failure.

int **security_inode_init_security**(struct *inode* *inode, struct *inode* *dir, const struct *qstr* *qstr, const *initxattrs* initxattrs, void *fs_data)

Initialize an inode's LSM context

Parameters

struct inode *inode
the inode

struct inode *dir
parent directory

const struct qstr *qstr
last component of the pathname

const initxattrs initxattrs
callback function to write xattrs

void *fs_data
filesystem specific data

Description

Obtain the security attribute name suffix and value to set on a newly created inode and set up the incore security field for the new inode. This hook is called by the fs code as part of the inode creation transaction and provides for atomic labeling of the inode, unlike the post_create/mkdir/... hooks called by the VFS.

The hook function is expected to populate the xattrs array, by calling lsm_get_xattr_slot() to retrieve the slots reserved by the security module with the lbs_xattr_count field of the

lsm_blob_sizes structure. For each slot, the hook function should set ->name to the attribute name suffix (e.g. selinux), to allocate ->value (will be freed by the caller) and set it to the attribute value, to set ->value_len to the length of the value. If the security module does not use security attributes or does not wish to put a security attribute on this particular inode, then it should return -EOPNOTSUPP to skip this processing.

Return

Returns 0 if the LSM successfully initialized all of the inode
security attributes that are required, negative values otherwise.

int **security_path_mknod**(const struct path *dir, struct *dentry* *dentry, umode_t mode, unsigned int dev)

Check if creating a special file is allowed

Parameters

const struct path *dir
parent directory

struct dentry *dentry
new file

umode_t mode
new file mode

unsigned int dev
device number

Description

Check permissions when creating a file. Note that this hook is called even if mknod operation is being done for a regular file.

Return

Returns 0 if permission is granted.

int **security_path_mkdir**(const struct path *dir, struct *dentry* *dentry, umode_t mode)

Check if creating a new directory is allowed

Parameters

const struct path *dir
parent directory

struct dentry *dentry
new directory

umode_t mode
new directory mode

Description

Check permissions to create a new directory in the existing directory.

Return

Returns 0 if permission is granted.

int **security_path_unlink**(const struct path *dir, struct *dentry* *dentry)
Check if removing a hard link is allowed

Parameters

const struct path *dir
parent directory

struct dentry *dentry
file

Description

Check the permission to remove a hard link to a file.

Return

Returns 0 if permission is granted.

int **security_path_rename**(const struct path *old_dir, struct dentry *old_dentry, const struct path *new_dir, struct dentry *new_dentry, unsigned int flags)
Check if renaming a file is allowed

Parameters

const struct path *old_dir
parent directory of the old file

struct dentry *old_dentry
the old file

const struct path *new_dir
parent directory of the new file

struct dentry *new_dentry
the new file

unsigned int flags
flags

Description

Check for permission to rename a file or directory.

Return

Returns 0 if permission is granted.

int **security_inode_create**(struct inode *dir, struct *dentry* *dentry, umode_t mode)
Check if creating a file is allowed

Parameters

struct inode *dir
the parent directory

struct dentry *dentry
the file being created

umode_t mode
requested file mode

Description

Check permission to create a regular file.

Return

Returns 0 if permission is granted.

int **security_inode_mkdir**(struct inode *dir, struct *dentry* *dentry, umode_t mode)
Check if creation a new director is allowed

Parameters

struct inode *dir
parent directory

struct dentry *dentry
new directory

umode_t mode
new directory mode

Description

Check permissions to create a new directory in the existing directory associated with inode structure **dir**.

Return

Returns 0 if permission is granted.

int **security_inode_setattr**(struct mnt_idmap *idmap, struct *dentry* *dentry, struct iattr *attr)
Check if setting file attributes is allowed

Parameters

struct mnt_idmap *idmap
idmap of the mount

struct dentry *dentry
file

struct iattr *attr
new attributes

Description

Check permission before setting file attributes. Note that the kernel call to notify_change is performed from several locations, whenever file attributes change (such as when a file is truncated, chown/chmod operations, transferring disk quotas, etc).

Return

Returns 0 if permission is granted.

int **security_inode_listsecurity**(struct *inode* *inode, char *buffer, size_t buffer_size)
List the xattr security label names

Parameters

struct inode *inode
inode

char *buffer
buffer

size_t buffer_size
size of buffer

Description

Copy the extended attribute names for the security labels associated with **inode** into **buffer**. The maximum size of **buffer** is specified by **buffer_size**. **buffer** may be NULL to request the size of the buffer required.

Return

Returns number of bytes used/required on success.

int **security_inode_copy_up**(struct dentry *src, struct cred **new)
Create new creds for an overlayfs copy-up op

Parameters

struct dentry *src
union dentry of copy-up file

struct cred **new
newly created creds

Description

A file is about to be copied up from lower layer to upper layer of overlay filesystem. Security module can prepare a set of new creds and modify as need be and return new creds. Caller will switch to new creds temporarily to create new file and release newly allocated creds.

Return

Returns 0 on success or a negative error code on error.

int **security_inode_copy_up_xattr**(const char *name)
Filter xattrs in an overlayfs copy-up op

Parameters

const char *name
xattr name

Description

Filter the xattrs being copied up when a unioned file is copied up from a lower layer to the union/overlay layer. The caller is responsible for reading and writing the xattrs, this hook is merely a filter.

Return

Returns 0 to accept the xattr, 1 to discard the xattr, -EOPNOTSUPP
if the security module does not know about attribute, or a negative error code to abort the copy up.

int **security_file_ioctl**(struct *file* *file, unsigned int cmd, unsigned long arg)
Check if an ioctl is allowed

Parameters

struct file *file
associated file

unsigned int cmd
ioctl cmd

unsigned long arg
ioctl arguments

Description

Check permission for an ioctl operation on **file**. Note that **arg** sometimes represents a user space pointer; in other cases, it may be a simple integer value. When **arg** represents a user space pointer, it should never be used by the security module.

Return

Returns 0 if permission is granted.

int **security_file_ioctl_compat**(struct *file* *file, unsigned int cmd, unsigned long arg)
Check if an ioctl is allowed in compat mode

Parameters

struct file *file
associated file

unsigned int cmd
ioctl cmd

unsigned long arg
ioctl arguments

Description

Compat version of *security_file_ioctl()* that correctly handles 32-bit processes running on 64-bit kernels.

Return

Returns 0 if permission is granted.

void **security_cred_getsecid**(const struct cred *c, u32 *secid)
Get the secid from a set of credentials

Parameters

const struct cred *c
credentials

u32 *secid
secid value

Description

Retrieve the security identifier of the cred structure **c**. In case of failure, **secid** will be set to zero.

int **security_kernel_read_file**(struct *file* *file, enum kernel_read_file_id id, bool contents)
Read a file specified by userspace

Parameters

struct file *file
file

enum kernel_read_file_id id
file identifier

bool contents
trust if *security_kernel_post_read_file()* will be called

Description

Read a file specified by userspace.

Return

Returns 0 if permission is granted.

int **security_kernel_post_read_file**(struct *file* *file, char *buf, loff_t size, enum kernel_read_file_id id)
Read a file specified by userspace

Parameters

struct file *file
file

char *buf
file contents

loff_t size
size of file contents

enum kernel_read_file_id id
file identifier

Description

Read a file specified by userspace. This must be paired with a prior call to *security_kernel_read_file()* call that indicated this hook would also be called, see *security_kernel_read_file()* for more information.

Return

Returns 0 if permission is granted.

int **security_kernel_load_data**(enum kernel_load_data_id id, bool contents)
Load data provided by userspace

Parameters

enum kernel_load_data_id id
data identifier

bool contents
true if *security_kernel_post_load_data()* will be called

Description

Load data provided by userspace.

Return

Returns 0 if permission is granted.

int **security_kernel_post_load_data**(char *buf, loff_t size, enum kernel_load_data_id id, char *description)

Load userspace data from a non-file source

Parameters

char *buf
data

loff_t size
size of data

enum kernel_load_data_id id
data identifier

char *description
text description of data, specific to the id value

Description

Load data provided by a non-file source (usually userspace buffer). This must be paired with a prior *security_kernel_load_data()* call that indicated this hook would also be called, see *security_kernel_load_data()* for more information.

Return

Returns 0 if permission is granted.

void **security_current_getsecid_subj**(u32 *secid)

Get the current task's subjective secid

Parameters

u32 *secid
secid value

Description

Retrieve the subjective security identifier of the current task and return it in **secid**. In case of failure, **secid** will be set to zero.

void **security_task_getsecid_obj**(struct task_struct *p, u32 *secid)

Get a task's objective secid

Parameters

struct task_struct *p
target task

u32 *secid
secid value

Description

Retrieve the objective security identifier of the task_struct in **p** and return it in **secid**. In case of failure, **secid** will be set to zero.

void **security_d_instantiate**(struct *dentry* *dentry, struct *inode* *inode)

Populate an inode's LSM state based on a dentry

Parameters

struct dentry *dentry
dentry

struct inode *inode
inode

Description

Fill in **inode** security information for a **dentry** if allowed.

int **security_ismaclabel**(const char *name)
Check is the named attribute is a MAC label

Parameters

const char *name
full extended attribute name

Description

Check if the extended attribute specified by **name** represents a MAC label.

Return

Returns 1 if name is a MAC attribute otherwise returns 0.

int **security_secid_to_secctx**(u32 secid, char **secdata, u32 *seclen)
Convert a secid to a secctx

Parameters

u32 secid
secid

char **secdata
secctx

u32 *seclen
secctx length

Description

Convert secid to security context. If **secdata** is NULL the length of the result will be returned in **seclen**, but no **secdata** will be returned. This does mean that the length could change between calls to check the length and the next call which actually allocates and returns the **secdata**.

Return

Return 0 on success, error on failure.

int **security_secctx_to_secid**(const char *secdata, u32 seclen, u32 *secid)
Convert a secctx to a secid

Parameters

const char *secdata
secctx

u32 seclen
length of secctx

u32 *secid
secid

Description

Convert security context to secid.

Return

Returns 0 on success, error on failure.

void **security_release_secctx**(char *secdata, u32 seclen)
Free a secctx buffer

Parameters

char *secdata
secctx

u32 seclen
length of secctx

Description

Release the security context.

void **security_inode_invalidate_secctx**(struct *inode* *inode)
Invalidate an inode's security label

Parameters

struct inode *inode
inode

Description

Notify the security module that it must revalidate the security context of an inode.

int **security_inode_notifysecctx**(struct *inode* *inode, void *ctx, u32 ctxlen)
Notify the LSM of an inode's security label

Parameters

struct inode *inode
inode

void *ctx
secctx

u32 ctxlen
length of secctx

Description

Notify the security module of what the security context of an inode should be. Initializes the incore security context managed by the security module for this inode. Example usage: NFS client invokes this hook to initialize the security context in its incore inode to the value provided by the server for the file when the server returned the file's attributes to the client. Must be called with inode->i_mutex locked.

Return

Returns 0 on success, error on failure.

int **security_inode_setsecctx**(struct *dentry* *dentry, void *ctx, u32 ctxlen)

Change the security label of an inode

Parameters

struct dentry *dentry
inode

void *ctx
secctx

u32 ctxlen
length of secctx

Description

Change the security context of an inode. Updates the incore security context managed by the security module and invokes the fs code as needed (via __vfs_setxattr_noperm) to update any backing xattrs that represent the context. Example usage: NFS server invokes this hook to change the security context in its incore inode and on the backing filesystem to a value provided by the client on a SETATTR operation. Must be called with inode->i_mutex locked.

Return

Returns 0 on success, error on failure.

int **security_inode_getsecctx**(struct *inode* *inode, void **ctx, u32 *ctxlen)

Get the security label of an inode

Parameters

struct inode *inode
inode

void **ctx
secctx

u32 *ctxlen
length of secctx

Description

On success, returns 0 and fills out **ctx** and **ctxlen** with the security context for the given **inode**.

Return

Returns 0 on success, error on failure.

int **security_unix_stream_connect**(struct *sock* *sock, struct *sock* *other, struct *sock* *newsk)

Check if a AF_UNIX stream is allowed

Parameters

struct sock *sock
originating sock

struct sock *other
peer sock

struct sock *newsk
new sock

Description

Check permissions before establishing a Unix domain stream connection between **sock** and **other**.

The **unix_stream_connect** and **unix_may_send** hooks were necessary because Linux provides an alternative to the conventional file name space for Unix domain sockets. Whereas binding and connecting to sockets in the file name space is mediated by the typical file permissions (and caught by the mknod and permission hooks in inode_security_ops), binding and connecting to sockets in the abstract name space is completely unmediated. Sufficient control of Unix domain sockets in the abstract name space isn't possible using only the socket layer hooks, since we need to know the actual target socket, which is not looked up until we are inside the af_unix code.

Return

Returns 0 if permission is granted.

int **security_unix_may_send**(struct socket *sock, struct socket *other)

Check if AF_UNIX socket can send datagrams

Parameters

struct socket *sock
originating sock

struct socket *other
peer sock

Description

Check permissions before connecting or sending datagrams from **sock** to **other**.

The **unix_stream_connect** and **unix_may_send** hooks were necessary because Linux provides an alternative to the conventional file name space for Unix domain sockets. Whereas binding and connecting to sockets in the file name space is mediated by the typical file permissions (and caught by the mknod and permission hooks in inode_security_ops), binding and connecting to sockets in the abstract name space is completely unmediated. Sufficient control of Unix domain sockets in the abstract name space isn't possible using only the socket layer hooks, since we need to know the actual target socket, which is not looked up until we are inside the af_unix code.

Return

Returns 0 if permission is granted.

int **security_socket_socketpair**(struct socket *socka, struct socket *sockb)

Check if creating a socketpair is allowed

Parameters

struct socket *socka
first socket

struct socket *sockb
second socket

Description

Check permissions before creating a fresh pair of sockets.

Return

Returns 0 if permission is granted and the connection was
established.

int **security_sock_rcv_skb**(struct sock *sk, struct sk_buff *skb)
Check if an incoming network packet is allowed

Parameters

struct sock *sk
destination sock

struct sk_buff *skb
incoming packet

Description

Check permissions on incoming network packets. This hook is distinct from Netfilter's IP input hooks since it is the first time that the incoming sk_buff **skb** has been associated with a particular socket, **sk**. Must not sleep inside this hook because some callers hold spinlocks.

Return

Returns 0 if permission is granted.

int **security_socket_getpeersec_dgram**(struct socket *sock, struct sk_buff *skb, u32 *secid)
Get the remote peer label

Parameters

struct socket *sock
socket

struct sk_buff *skb
datagram packet

u32 *secid
remote peer label secid

Description

This hook allows the security module to provide peer socket security state for udp sockets on a per-packet basis to userspace via getsockopt SO_GETPEERSEC. The application must first have indicated the IP_PASSSEC option via getsockopt. It can then retrieve the security state returned by this hook for a packet via the SCM_SECURITY ancillary message type.

Return

Returns 0 on success, error on failure.

void **security_sk_clone**(const struct sock *sk, struct sock *newsk)
Clone a sock's LSM state

Parameters

const struct sock *sk
original sock

struct sock *newsk
target sock

Description

Clone/copy security structure.

void **security_sk_classify_flow**(const struct sock *sk, struct flowi_common *flic)

Set a flow's secid based on socket

Parameters

const struct sock *sk
original socket

struct flowi_common *flic
target flow

Description

Set the target flow's secid to socket's secid.

void **security_req_classify_flow**(const struct request_sock *req, struct flowi_common *flic)

Set a flow's secid based on request_sock

Parameters

const struct request_sock *req
request_sock

struct flowi_common *flic
target flow

Description

Sets **flic**'s secid to **req**'s secid.

void **security_sock_graft**(struct sock *sk, struct socket *parent)

Reconcile LSM state when grafting a sock on a socket

Parameters

struct sock *sk
sock being grafted

struct socket *parent
target parent socket

Description

Sets **parent**'s inode secid to **sk**'s secid and update **sk** with any necessary LSM state from **parent**.

int **security_inet_conn_request**(const struct sock *sk, struct sk_buff *skb, struct request_sock *req)

Set request_sock state using incoming connect

Parameters

const struct sock *sk
parent listening sock

struct sk_buff *skb
incoming connection

struct request_sock *req
new request_sock

Description

Initialize the **req** LSM state based on **sk** and the incoming connect in **skb**.

Return

Returns 0 if permission is granted.

void **security_inet_conn_established**(struct sock *sk, struct sk_buff *skb)
Update sock's LSM state with connection

Parameters

struct sock *sk
sock

struct sk_buff *skb
connection packet

Description

Update **sock**'s LSM state to represent a new connection from **skb**.

int **security_secmark_relabel_packet**(u32 secid)
Check if setting a secmark is allowed

Parameters

u32 secid
new secmark value

Description

Check if the process should be allowed to relabel packets to **secid**.

Return

Returns 0 if permission is granted.

void **security_secmark_refcount_inc**(void)
Increment the secmark labeling rule count

Parameters

void
no arguments

Description

Tells the LSM to increment the number of secmark labeling rules loaded.

void **security_secmark_refcount_dec**(void)
Decrement the secmark labeling rule count

Parameters

void
no arguments

Description

Tells the LSM to decrement the number of secmark labeling rules loaded.

int **security_tun_dev_alloc_security**(void **security)

Allocate a LSM blob for a TUN device

Parameters

void **security

pointer to the LSM blob

Description

This hook allows a module to allocate a security structure for a TUN device, returning the pointer in **security**.

Return

Returns a zero on success, negative values on failure.

void **security_tun_dev_free_security**(void *security)

Free a TUN device LSM blob

Parameters

void *security

LSM blob

Description

This hook allows a module to free the security structure for a TUN device.

int **security_tun_dev_create**(void)

Check if creating a TUN device is allowed

Parameters

void

no arguments

Description

Check permissions prior to creating a new TUN device.

Return

Returns 0 if permission is granted.

int **security_tun_dev_attach_queue**(void *security)

Check if attaching a TUN queue is allowed

Parameters

void *security

TUN device LSM blob

Description

Check permissions prior to attaching to a TUN device queue.

Return

Returns 0 if permission is granted.

int **security_tun_dev_attach**(struct sock *sk, void *security)

Update TUN device LSM state on attach

Parameters

struct sock *sk
associated sock

void *security
TUN device LSM blob

Description

This hook can be used by the module to update any security state associated with the TUN device's sock structure.

Return

Returns 0 if permission is granted.

int **security_tun_dev_open**(void *security)

Update TUN device LSM state on open

Parameters

void *security
TUN device LSM blob

Description

This hook can be used by the module to update any security state associated with the TUN device's security structure.

Return

Returns 0 if permission is granted.

int **security_sctp_assoc_request**(struct sctp_association *asoc, struct sk_buff *skb)

Update the LSM on a SCTP association req

Parameters

struct sctp_association *asoc
SCTP association

struct sk_buff *skb
packet requesting the association

Description

Passes the **asoc** and **chunk->skb** of the association INIT packet to the LSM.

Return

Returns 0 on success, error on failure.

int **security_sctp_bind_connect**(struct sock *sk, int optname, struct sockaddr *address, int addrlen)

Validate a list of addrs for a SCTP option

Parameters

struct sock *sk
socket

int optname
SCTP option to validate

struct sockaddr *address
list of IP addresses to validate

int addrlen
length of the address list

Description

Validiate permissions required for each address associated with sock **sk**. Depending on **optname**, the addresses will be treated as either a connect or bind service. The **addrlen** is calculated on each IPv4 and IPv6 address using sizeof(struct sockaddr_in) or sizeof(struct sockaddr_in6).

Return

Returns 0 on success, error on failure.

void **security_sctp_sk_clone**(struct sctp_association *asoc, struct sock *sk, struct sock *newsk)

Clone a SCTP sock's LSM state

Parameters

struct sctp_association *asoc
SCTP association

struct sock *sk
original sock

struct sock *newsk
target sock

Description

Called whenever a new socket is created by accept(2) (i.e. a TCP style socket) or when a socket is 'peeled off' e.g userspace calls sctp_peeloff(3).

int **security_sctp_assoc_established**(struct sctp_association *asoc, struct sk_buff *skb)

Update LSM state when assoc established

Parameters

struct sctp_association *asoc
SCTP association

struct sk_buff *skb
packet establishing the association

Description

Passes the **asoc** and **chunk->skb** of the association COOKIE_ACK packet to the security module.

Return

Returns 0 if permission is granted.

int **security_ib_pkey_access**(void *sec, u64 subnet_prefix, u16 pkey)
Check if access to an IB pkey is allowed

Parameters

void *sec
LSM blob

u64 subnet_prefix
subnet prefix of the port

u16 pkey
IB pkey

Description

Check permission to access a pkey when modifying a QP.

Return

Returns 0 if permission is granted.

int **security_ib_endport_manage_subnet**(void *sec, const char *dev_name, u8 port_num)
Check if SMPs traffic is allowed

Parameters

void *sec
LSM blob

const char *dev_name
IB device name

u8 port_num
port number

Description

Check permissions to send and receive SMPs on a end port.

Return

Returns 0 if permission is granted.

int **security_ib_alloc_security**(void **sec)
Allocate an Infiniband LSM blob

Parameters

void **sec
LSM blob

Description

Allocate a security structure for Infiniband objects.

Return

Returns 0 on success, non-zero on failure.

void **security_ib_free_security**(void *sec)
Free an Infiniband LSM blob

Parameters

void *sec
LSM blob

Description

Deallocate an Infiniband security structure.

int **security_xfrm_policy_alloc**(struct xfrm_sec_ctx **ctxp, struct xfrm_user_sec_ctx *sec_ctx, gfp_t gfp)

Allocate a xfrm policy LSM blob

Parameters

struct xfrm_sec_ctx **ctxp
xfrm security context being added to the SPD

struct xfrm_user_sec_ctx *sec_ctx
security label provided by userspace

gfp_t gfp
gfp flags

Description

Allocate a security structure to the xp->security field; the security field is initialized to NULL when the xfrm_policy is allocated.

Return

Return 0 if operation was successful.

void **security_xfrm_policy_free**(struct xfrm_sec_ctx *ctx)

Free a xfrm security context

Parameters

struct xfrm_sec_ctx *ctx
xfrm security context

Description

Free LSM resources associated with **ctx**.

int **security_xfrm_state_alloc**(struct xfrm_state *x, struct xfrm_user_sec_ctx *sec_ctx)

Allocate a xfrm state LSM blob

Parameters

struct xfrm_state *x
xfrm state being added to the SAD

struct xfrm_user_sec_ctx *sec_ctx
security label provided by userspace

Description

Allocate a security structure to the **x->security** field; the security field is initialized to NULL when the xfrm_state is allocated. Set the context to correspond to **sec_ctx**.

Return

Return 0 if operation was successful.

int **security_xfrm_state_delete**(struct xfrm_state *x)
Check if deleting a xfrm state is allowed

Parameters

struct xfrm_state *x
xfrm state

Description

Authorize deletion of x->security.

Return

Returns 0 if permission is granted.

int **security_locked_down**(enum lockdown_reason what)
Check if a kernel feature is allowed

Parameters

enum lockdown_reason what
requested kernel feature

Description

Determine whether a kernel feature that potentially enables arbitrary code execution in kernel space should be permitted.

Return

Returns 0 if permission is granted.

CHAPTER

SEVEN

LINUX SECURE ATTENTION KEY (SAK) HANDLING

Date
18 March 2001

Author
Andrew Morton

An operating system's Secure Attention Key is a security tool which is provided as protection against trojan password capturing programs. It is an undefeatable way of killing all programs which could be masquerading as login applications. Users need to be taught to enter this key sequence before they log in to the system.

From the PC keyboard, Linux has two similar but different ways of providing SAK. One is the ALT-SYSRQ-K sequence. You shouldn't use this sequence. It is only available if the kernel was compiled with sysrq support.

The proper way of generating a SAK is to define the key sequence using `loadkeys`. This will work whether or not sysrq support is compiled into the kernel.

SAK works correctly when the keyboard is in raw mode. This means that once defined, SAK will kill a running X server. If the system is in run level 5, the X server will restart. This is what you want to happen.

What key sequence should you use? Well, CTRL-ALT-DEL is used to reboot the machine. CTRL-ALT-BACKSPACE is magical to the X server. We'll choose CTRL-ALT-PAUSE.

In your rc.sysinit (or rc.local) file, add the command:

```
echo "control alt keycode 101 = SAK" | /bin/loadkeys
```

And that's it! Only the superuser may reprogram the SAK key.

Note:

1. Linux SAK is said to be not a ”true SAK” as is required by systems which implement C2 level security. This author does not know why.

2. On the PC keyboard, SAK kills all applications which have /dev/console opened.

   Unfortunately this includes a number of things which you don't actually want killed. This is because these applications are incorrectly holding /dev/console open. Be sure to complain to your Linux distributor about this!

   You can identify processes which will be killed by SAK with the command:

```
# ls -l /proc/[0-9]*/fd/* | grep console
l-wx------    1 root     root           64 Mar 18 00:46 /proc/579/fd/0 -> /
 ↪dev/console
```

Then:

```
# ps aux|grep 579
root       579  0.0  0.1  1088  436 ?        S    00:43   0:00 gpm -t ps/2
```

So gpm will be killed by SAK. This is a bug in gpm. It should be closing standard input. You can work around this by finding the initscript which launches gpm and changing it thusly:

Old:

```
daemon gpm
```

New:

```
daemon gpm < /dev/null
```

Vixie cron also seems to have this problem, and needs the same treatment.

Also, one prominent Linux distribution has the following three lines in its rc.sysinit and rc scripts:

```
exec 3<&0
exec 4>&1
exec 5>&2
```

These commands cause **all** daemons which are launched by the initscripts to have file descriptors 3, 4 and 5 attached to /dev/console. So SAK kills them all. A workaround is to simply delete these lines, but this may cause system management applications to malfunction - test everything well.

CHAPTER

EIGHT

SCTP

8.1 SCTP LSM Support

8.1.1 Security Hooks

For security module support, three SCTP specific hooks have been implemented:

```
security_sctp_assoc_request()
security_sctp_bind_connect()
security_sctp_sk_clone()
security_sctp_assoc_established()
```

The usage of these hooks are described below with the SELinux implementation described in the *SCTP SELinux Support* chapter.

security_sctp_assoc_request()

Passes the `@asoc` and `@chunk->skb` of the association INIT packet to the security module. Returns 0 on success, error on failure.

```
@asoc - pointer to sctp association structure.
@skb - pointer to skbuff of association packet.
```

security_sctp_bind_connect()

Passes one or more ipv4/ipv6 addresses to the security module for validation based on the `@optname` that will result in either a bind or connect service as shown in the permission check tables below. Returns 0 on success, error on failure.

```
  @sk      - Pointer to sock structure.
  @optname - Name of the option to validate.
  @address - One or more ipv4 / ipv6 addresses.
  @addrlen - The total length of address(s). This is calculated on each
             ipv4 or ipv6 address using sizeof(struct sockaddr_in) or
             sizeof(struct sockaddr_in6).

-------------------------------------------------------------------
|                     BIND Type Checks                            |
```

```
|       @optname              |         @address contains          |
|-----------------------------|------------------------------------|
| SCTP_SOCKOPT_BINDX_ADD      | One or more ipv4 / ipv6 addresses |
| SCTP_PRIMARY_ADDR           | Single ipv4 or ipv6 address       |
| SCTP_SET_PEER_PRIMARY_ADDR | Single ipv4 or ipv6 address       |
-------------------------------------------------------------------


-------------------------------------------------------------------
|                    CONNECT Type Checks                           |
|       @optname              |         @address contains          |
|-----------------------------|------------------------------------|
| SCTP_SOCKOPT_CONNECTX       | One or more ipv4 / ipv6 addresses |
| SCTP_PARAM_ADD_IP           | One or more ipv4 / ipv6 addresses |
| SCTP_SENDMSG_CONNECT        | Single ipv4 or ipv6 address       |
| SCTP_PARAM_SET_PRIMARY      | Single ipv4 or ipv6 address       |
-------------------------------------------------------------------
```

A summary of the `@optname` entries is as follows:

```
SCTP_SOCKOPT_BINDX_ADD - Allows additional bind addresses to be
                         associated after (optionally) calling
                         bind(3).
                         sctp_bindx(3) adds a set of bind
                         addresses on a socket.

SCTP_SOCKOPT_CONNECTX - Allows the allocation of multiple
                        addresses for reaching a peer
                        (multi-homed).
                        sctp_connectx(3) initiates a connection
                        on an SCTP socket using multiple
                        destination addresses.

SCTP_SENDMSG_CONNECT  - Initiate a connection that is generated by a
                        sendmsg(2) or sctp_sendmsg(3) on a new asociation.

SCTP_PRIMARY_ADDR     - Set local primary address.

SCTP_SET_PEER_PRIMARY_ADDR - Request peer sets address as
                             association primary.

SCTP_PARAM_ADD_IP          - These are used when Dynamic Address
SCTP_PARAM_SET_PRIMARY     - Reconfiguration is enabled as explained below.
```

To support Dynamic Address Reconfiguration the following parameters must be enabled on both endpoints (or use the appropriate **setsockopt**(2)):

```
/proc/sys/net/sctp/addip_enable
/proc/sys/net/sctp/addip_noauth_enable
```

then the following *_PARAM_*'s are sent to the peer in an ASCONF chunk when the corresponding `@optname`'s are present:

```
      @optname                      ASCONF Parameter
     ----------                    ------------------
SCTP_SOCKOPT_BINDX_ADD     ->   SCTP_PARAM_ADD_IP
SCTP_SET_PEER_PRIMARY_ADDR ->   SCTP_PARAM_SET_PRIMARY
```

security_sctp_sk_clone()

Called whenever a new socket is created by **accept**(2) (i.e. a TCP style socket) or when a socket is 'peeled off' e.g userspace calls **sctp_peeloff**(3).

```
@asoc - pointer to current sctp association structure.
@sk - pointer to current sock structure.
@newsk - pointer to new sock structure.
```

security_sctp_assoc_established()

Called when a COOKIE ACK is received, and the peer secid will be saved into `@asoc->peer_secid` for client:

```
@asoc - pointer to sctp association structure.
@skb - pointer to skbuff of the COOKIE ACK packet.
```

8.1.2 Security Hooks used for Association Establishment

The following diagram shows the use of `security_sctp_bind_connect()`, `security_sctp_assoc_request()`, `security_sctp_assoc_established()` when establishing an association.

```
     SCTP endpoint "A"                                    SCTP endpoint "Z"
     =================                                    =================
   sctp_sf_do_prm_asoc()
Association setup can be initiated
by a connect(2), sctp_connectx(3),
sendmsg(2) or sctp_sendmsg(3).
These will result in a call to
security_sctp_bind_connect() to
initiate an association to
SCTP peer endpoint "Z".
        INIT ------------------------------------------->
                                                  sctp_sf_do_5_1B_init()
                                               Respond to an INIT chunk.
                                          SCTP peer endpoint "A" is asking
                                          for a temporary association.
                                          Call security_sctp_assoc_request()
                                          to set the peer label if first
                                          association.
                                          If not first association, check
                                          whether allowed, IF so send:
```

```
        <---------------------------------------------- INIT ACK
        |                                    ELSE audit event and silently
        |                                         discard the packet.
        |
   COOKIE ECHO ----------------------------------------->
                                                sctp_sf_do_5_1D_ce()
                                           Respond to an COOKIE ECHO chunk.
                                           Confirm the cookie and create a
                                           permanent association.
                                           Call security_sctp_assoc_request()␣
↪to
                                           do the same as for INIT chunk␣
↪Response.
        <---------------------------------------------- COOKIE ACK
        |                                                    |
   sctp_sf_do_5_1E_ca                                        |
Call security_sctp_assoc_established()                       |
to set the peer label.                                       |
        |                                                    |
        |                                  If SCTP_SOCKET_TCP or peeled off
        |                                  socket security_sctp_sk_clone() is
        |                                  called to clone the new socket.
        |                                                    |
     ESTABLISHED                                        ESTABLISHED
        |                                                    |
   ------------------------------------------------------------------------
   |                      Association Established                          |
   ------------------------------------------------------------------------
```

8.2 SCTP SELinux Support

8.2.1 Security Hooks

The *SCTP LSM Support* chapter above describes the following SCTP security hooks with the SELinux specifics expanded below:

```
security_sctp_assoc_request()
security_sctp_bind_connect()
security_sctp_sk_clone()
security_sctp_assoc_established()
```

security_sctp_assoc_request()

Passes the `@asoc` and `@chunk->skb` of the association INIT packet to the security module. Returns 0 on success, error on failure.

```
@asoc - pointer to sctp association structure.
@skb - pointer to skbuff of association packet.
```

The security module performs the following operations:

IF this is the first association on `@asoc->base.sk`, then set the peer sid to that in `@skb`. This will ensure there is only one peer sid assigned to `@asoc->base.sk` that may support multiple associations.

ELSE validate the `@asoc->base.sk peer_sid` against the `@skb peer sid` to determine whether the association should be allowed or denied.

Set the sctp `@asoc sid` to socket's sid (from `asoc->base.sk`) with MLS portion taken from `@skb peer sid`. This will be used by SCTP TCP style sockets and peeled off connections as they cause a new socket to be generated.

If IP security options are configured (CIPSO/CALIPSO), then the ip options are set on the socket.

security_sctp_bind_connect()

Checks permissions required for ipv4/ipv6 addresses based on the `@optname` as follows:

```
-----------------------------------------------------------------
|                    BIND Permission Checks                     |
|       @optname              |         @address contains       |
|-----------------------------|---------------------------------|
| SCTP_SOCKOPT_BINDX_ADD      | One or more ipv4 / ipv6 addresses |
| SCTP_PRIMARY_ADDR           | Single ipv4 or ipv6 address       |
| SCTP_SET_PEER_PRIMARY_ADDR | Single ipv4 or ipv6 address       |
-----------------------------------------------------------------

-----------------------------------------------------------------
|                  CONNECT Permission Checks                    |
|       @optname              |         @address contains       |
|-----------------------------|---------------------------------|
| SCTP_SOCKOPT_CONNECTX       | One or more ipv4 / ipv6 addresses |
| SCTP_PARAM_ADD_IP           | One or more ipv4 / ipv6 addresses |
| SCTP_SENDMSG_CONNECT        | Single ipv4 or ipv6 address       |
| SCTP_PARAM_SET_PRIMARY      | Single ipv4 or ipv6 address       |
-----------------------------------------------------------------
```

SCTP LSM Support gives a summary of the `@optname` entries and also describes ASCONF chunk processing when Dynamic Address Reconfiguration is enabled.

security_sctp_sk_clone()

Called whenever a new socket is created by **accept**(2) (i.e. a TCP style socket) or when a socket is 'peeled off' e.g userspace calls **sctp_peeloff**(3). `security_sctp_sk_clone()` will set the new sockets sid and peer sid to that contained in the `@asoc sid` and `@asoc peer sid` respectively.

```
@asoc - pointer to current sctp association structure.
@sk - pointer to current sock structure.
@newsk - pointer to new sock structure.
```

security_sctp_assoc_established()

Called when a COOKIE ACK is received where it sets the connection's peer sid to that in `@skb`:

```
@asoc - pointer to sctp association structure.
@skb - pointer to skbuff of the COOKIE ACK packet.
```

8.2.2 Policy Statements

The following class and permissions to support SCTP are available within the kernel:

```
class sctp_socket inherits socket { node_bind }
```

whenever the following policy capability is enabled:

```
policycap extended_socket_class;
```

SELinux SCTP support adds the `name_connect` permission for connecting to a specific port type and the `association` permission that is explained in the section below.

If userspace tools have been updated, SCTP will support the `portcon` statement as shown in the following example:

```
portcon sctp 1024-1036 system_u:object_r:sctp_ports_t:s0
```

8.2.3 SCTP Peer Labeling

An SCTP socket will only have one peer label assigned to it. This will be assigned during the establishment of the first association. Any further associations on this socket will have their packet peer label compared to the sockets peer label, and only if they are different will the `association` permission be validated. This is validated by checking the socket peer sid against the received packets peer sid to determine whether the association should be allowed or denied.

NOTES:

1) If peer labeling is not enabled, then the peer context will always be `SECINITSID_UNLABELED` (`unlabeled_t` in Reference Policy).

2) As SCTP can support more than one transport address per endpoint (multi-homing) on a single socket, it is possible to configure policy and NetLabel to provide different peer

labels for each of these. As the socket peer label is determined by the first associations transport address, it is recommended that all peer labels are consistent.

3) **getpeercon**(3) may be used by userspace to retrieve the sockets peer context.

4) While not SCTP specific, be aware when using NetLabel that if a label is assigned to a specific interface, and that interface 'goes down', then the NetLabel service will remove the entry. Therefore ensure that the network startup scripts call **netlabelctl**(8) to set the required label (see **netlabel-config**(8) helper script for details).

5) The NetLabel SCTP peer labeling rules apply as discussed in the following set of posts tagged ”netlabel” at: https://www.paul-moore.com/blog/t.

6) CIPSO is only supported for IPv4 addressing: `socket(AF_INET, ...)` CALIPSO is only supported for IPv6 addressing: `socket(AF_INET6, ...)`

   Note the following when testing CIPSO/CALIPSO:

   a) CIPSO will send an ICMP packet if an SCTP packet cannot be delivered because of an invalid label.

   b) CALIPSO does not send an ICMP packet, just silently discards it.

7) IPSEC is not supported as RFC 3554 - sctp/ipsec support has not been implemented in userspace (**racoon**(8) or **ipsec_pluto**(8)), although the kernel supports SCTP/IPSEC.

CHAPTER

NINE

KERNEL SELF-PROTECTION

Kernel self-protection is the design and implementation of systems and structures within the Linux kernel to protect against security flaws in the kernel itself. This covers a wide range of issues, including removing entire classes of bugs, blocking security flaw exploitation methods, and actively detecting attack attempts. Not all topics are explored in this document, but it should serve as a reasonable starting point and answer any frequently asked questions. (Patches welcome, of course!)

In the worst-case scenario, we assume an unprivileged local attacker has arbitrary read and write access to the kernel's memory. In many cases, bugs being exploited will not provide this level of access, but with systems in place that defend against the worst case we'll cover the more limited cases as well. A higher bar, and one that should still be kept in mind, is protecting the kernel against a _privileged_ local attacker, since the root user has access to a vastly increased attack surface. (Especially when they have the ability to load arbitrary kernel modules.)

The goals for successful self-protection systems would be that they are effective, on by default, require no opt-in by developers, have no performance impact, do not impede kernel debugging, and have tests. It is uncommon that all these goals can be met, but it is worth explicitly mentioning them, since these aspects need to be explored, dealt with, and/or accepted.

9.1 Attack Surface Reduction

The most fundamental defense against security exploits is to reduce the areas of the kernel that can be used to redirect execution. This ranges from limiting the exposed APIs available to userspace, making in-kernel APIs hard to use incorrectly, minimizing the areas of writable kernel memory, etc.

9.1.1 Strict kernel memory permissions

When all of kernel memory is writable, it becomes trivial for attacks to redirect execution flow. To reduce the availability of these targets the kernel needs to protect its memory with a tight set of permissions.

Executable code and read-only data must not be writable

Any areas of the kernel with executable memory must not be writable. While this obviously includes the kernel text itself, we must consider all additional places too: kernel modules, JIT memory, etc. (There are temporary exceptions to this rule to support things like instruction alternatives, breakpoints, kprobes, etc. If these must exist in a kernel, they are implemented in a way where the memory is temporarily made writable during the update, and then returned to the original permissions.)

In support of this are `CONFIG_STRICT_KERNEL_RWX` and `CONFIG_STRICT_MODULE_RWX`, which seek to make sure that code is not writable, data is not executable, and read-only data is neither writable nor executable.

Most architectures have these options on by default and not user selectable. For some architectures like arm that wish to have these be selectable, the architecture Kconfig can select ARCH_OPTIONAL_KERNEL_RWX to enable a Kconfig prompt. `CONFIG_ARCH_OPTIONAL_KERNEL_RWX_DEFAULT` determines the default setting when ARCH_OPTIONAL_KERNEL_RWX is enabled.

Function pointers and sensitive variables must not be writable

Vast areas of kernel memory contain function pointers that are looked up by the kernel and used to continue execution (e.g. descriptor/vector tables, file/network/etc operation structures, etc). The number of these variables must be reduced to an absolute minimum.

Many such variables can be made read-only by setting them ”const” so that they live in the .rodata section instead of the .data section of the kernel, gaining the protection of the kernel's strict memory permissions as described above.

For variables that are initialized once at `__init` time, these can be marked with the `__ro_after_init` attribute.

What remains are variables that are updated rarely (e.g. GDT). These will need another infrastructure (similar to the temporary exceptions made to kernel code mentioned above) that allow them to spend the rest of their lifetime read-only. (For example, when being updated, only the CPU thread performing the update would be given uninterruptible write access to the memory.)

Segregation of kernel memory from userspace memory

The kernel must never execute userspace memory. The kernel must also never access userspace memory without explicit expectation to do so. These rules can be enforced either by support of hardware-based restrictions (x86's SMEP/SMAP, ARM's PXN/PAN) or via emulation (ARM's Memory Domains). By blocking userspace memory in this way, execution and data parsing cannot be passed to trivially-controlled userspace memory, forcing attacks to operate entirely in kernel memory.

9.1.2 Reduced access to syscalls

One trivial way to eliminate many syscalls for 64-bit systems is building without `CONFIG_COMPAT`. However, this is rarely a feasible scenario.

The ”seccomp” system provides an opt-in feature made available to userspace, which provides a way to reduce the number of kernel entry points available to a running process. This limits the breadth of kernel code that can be reached, possibly reducing the availability of a given bug to an attack.

An area of improvement would be creating viable ways to keep access to things like compat, user namespaces, BPF creation, and perf limited only to trusted processes. This would keep the scope of kernel entry points restricted to the more regular set of normally available to unprivileged userspace.

9.1.3 Restricting access to kernel modules

The kernel should never allow an unprivileged user the ability to load specific kernel modules, since that would provide a facility to unexpectedly extend the available attack surface. (The on-demand loading of modules via their predefined subsystems, e.g. MODULE_ALIAS_*, is considered ”expected” here, though additional consideration should be given even to these.) For example, loading a filesystem module via an unprivileged socket API is nonsense: only the root or physically local user should trigger filesystem module loading. (And even this can be up for debate in some scenarios.)

To protect against even privileged users, systems may need to either disable module loading entirely (e.g. monolithic kernel builds or modules_disabled sysctl), or provide signed modules (e.g. `CONFIG_MODULE_SIG_FORCE`, or dm-crypt with LoadPin), to keep from having root load arbitrary kernel code via the module loader interface.

9.2 Memory integrity

There are many memory structures in the kernel that are regularly abused to gain execution control during an attack, By far the most commonly understood is that of the stack buffer overflow in which the return address stored on the stack is overwritten. Many other examples of this kind of attack exist, and protections exist to defend against them.

9.2.1 Stack buffer overflow

The classic stack buffer overflow involves writing past the expected end of a variable stored on the stack, ultimately writing a controlled value to the stack frame's stored return address. The most widely used defense is the presence of a stack canary between the stack variables and the return address (`CONFIG_STACKPROTECTOR`), which is verified just before the function returns. Other defenses include things like shadow stacks.

9.2.2 Stack depth overflow

A less well understood attack is using a bug that triggers the kernel to consume stack memory with deep function calls or large stack allocations. With this attack it is possible to write beyond the end of the kernel's preallocated stack space and into sensitive structures. Two important changes need to be made for better protections: moving the sensitive thread_info structure elsewhere, and adding a faulting memory hole at the bottom of the stack to catch these overflows.

9.2.3 Heap memory integrity

The structures used to track heap free lists can be sanity-checked during allocation and freeing to make sure they aren't being used to manipulate other memory areas.

9.2.4 Counter integrity

Many places in the kernel use atomic counters to track object references or perform similar lifetime management. When these counters can be made to wrap (over or under) this traditionally exposes a use-after-free flaw. By trapping atomic wrapping, this class of bug vanishes.

9.2.5 Size calculation overflow detection

Similar to counter overflow, integer overflows (usually size calculations) need to be detected at runtime to kill this class of bug, which traditionally leads to being able to write past the end of kernel buffers.

9.3 Probabilistic defenses

While many protections can be considered deterministic (e.g. read-only memory cannot be written to), some protections provide only statistical defense, in that an attack must gather enough information about a running system to overcome the defense. While not perfect, these do provide meaningful defenses.

9.3.1 Canaries, blinding, and other secrets

It should be noted that things like the stack canary discussed earlier are technically statistical defenses, since they rely on a secret value, and such values may become discoverable through an information exposure flaw.

Blinding literal values for things like JITs, where the executable contents may be partially under the control of userspace, need a similar secret value.

It is critical that the secret values used must be separate (e.g. different canary per stack) and high entropy (e.g. is the RNG actually working?) in order to maximize their success.

9.3.2 Kernel Address Space Layout Randomization (KASLR)

Since the location of kernel memory is almost always instrumental in mounting a successful attack, making the location non-deterministic raises the difficulty of an exploit. (Note that this in turn makes the value of information exposures higher, since they may be used to discover desired memory locations.)

Text and module base

By relocating the physical and virtual base address of the kernel at boot-time (`CONFIG_RANDOMIZE_BASE`), attacks needing kernel code will be frustrated. Additionally, offsetting the module loading base address means that even systems that load the same set of modules in the same order every boot will not share a common base address with the rest of the kernel text.

Stack base

If the base address of the kernel stack is not the same between processes, or even not the same between syscalls, targets on or beyond the stack become more difficult to locate.

Dynamic memory base

Much of the kernel's dynamic memory (e.g. kmalloc, vmalloc, etc) ends up being relatively deterministic in layout due to the order of early-boot initializations. If the base address of these areas is not the same between boots, targeting them is frustrated, requiring an information exposure specific to the region.

Structure layout

By performing a per-build randomization of the layout of sensitive structures, attacks must either be tuned to known kernel builds or expose enough kernel memory to determine structure layouts before manipulating them.

9.4 Preventing Information Exposures

Since the locations of sensitive structures are the primary target for attacks, it is important to defend against exposure of both kernel memory addresses and kernel memory contents (since they may contain kernel addresses or other sensitive things like canary values).

9.4.1 Kernel addresses

Printing kernel addresses to userspace leaks sensitive information about the kernel memory layout. Care should be exercised when using any printk specifier that prints the raw address, currently %px, %p[ad], (and %p[sSb] in certain circumstances [*]). Any file written to using one of these specifiers should be readable only by privileged processes.

Kernels 4.14 and older printed the raw address using %p. As of 4.15-rc1 addresses printed with the specifier %p are hashed before printing.

[*] If KALLSYMS is enabled and symbol lookup fails, the raw address is printed. If KALLSYMS is not enabled the raw address is printed.

9.4.2 Unique identifiers

Kernel memory addresses must never be used as identifiers exposed to userspace. Instead, use an atomic counter, an idr, or similar unique identifier.

9.4.3 Memory initialization

Memory copied to userspace must always be fully initialized. If not explicitly memset(), this will require changes to the compiler to make sure structure holes are cleared.

9.4.4 Memory poisoning

When releasing memory, it is best to poison the contents, to avoid reuse attacks that rely on the old contents of memory. E.g., clear stack on a syscall return (`CONFIG_GCC_PLUGIN_STACKLEAK`), wipe heap memory on a free. This frustrates many uninitialized variable attacks, stack content exposures, heap content exposures, and use-after-free attacks.

9.4.5 Destination tracking

To help kill classes of bugs that result in kernel addresses being written to userspace, the destination of writes needs to be tracked. If the buffer is destined for userspace (e.g. seq_file backed `/proc` files), it should automatically censor sensitive values.

CHAPTER

TEN

SIPHASH - A SHORT INPUT PRF

Author

Written by Jason A. Donenfeld <jason@zx2c4.com>

SipHash is a cryptographically secure PRF -- a keyed hash function -- that performs very well for short inputs, hence the name. It was designed by cryptographers Daniel J. Bernstein and Jean-Philippe Aumasson. It is intended as a replacement for some uses of: *jhash*, *md5_transform*, *sha1_transform*, and so forth.

SipHash takes a secret key filled with randomly generated numbers and either an input buffer or several input integers. It spits out an integer that is indistinguishable from random. You may then use that integer as part of secure sequence numbers, secure cookies, or mask it off for use in a hash table.

10.1 Generating a key

Keys should always be generated from a cryptographically secure source of random numbers, either using get_random_bytes or get_random_once:

```
siphash_key_t key;
get_random_bytes(&key, sizeof(key));
```

If you're not deriving your key from here, you're doing it wrong.

10.2 Using the functions

There are two variants of the function, one that takes a list of integers, and one that takes a buffer:

```
u64 siphash(const void *data, size_t len, const siphash_key_t *key);
```

And:

```
u64 siphash_1u64(u64, const siphash_key_t *key);
u64 siphash_2u64(u64, u64, const siphash_key_t *key);
u64 siphash_3u64(u64, u64, u64, const siphash_key_t *key);
u64 siphash_4u64(u64, u64, u64, u64, const siphash_key_t *key);
u64 siphash_1u32(u32, const siphash_key_t *key);
u64 siphash_2u32(u32, u32, const siphash_key_t *key);
```

```
u64 siphash_3u32(u32, u32, u32, const siphash_key_t *key);
u64 siphash_4u32(u32, u32, u32, u32, const siphash_key_t *key);
```

If you pass the generic siphash function something of a constant length, it will constant fold at compile-time and automatically choose one of the optimized functions.

Hashtable key function usage:

```
struct some_hashtable {
        DECLARE_HASHTABLE(hashtable, 8);
        siphash_key_t key;
};

void init_hashtable(struct some_hashtable *table)
{
        get_random_bytes(&table->key, sizeof(table->key));
}

static inline hlist_head *some_hashtable_bucket(struct some_hashtable *table,␣
↪struct interesting_input *input)
{
        return &table->hashtable[siphash(input, sizeof(*input), &table->key) &␣
↪(HASH_SIZE(table->hashtable) - 1)];
}
```

You may then iterate like usual over the returned hash bucket.

10.3 Security

SipHash has a very high security margin, with its 128-bit key. So long as the key is kept secret, it is impossible for an attacker to guess the outputs of the function, even if being able to observe many outputs, since 2^128 outputs is significant.

Linux implements the ”2-4” variant of SipHash.

10.4 Struct-passing Pitfalls

Often times the XuY functions will not be large enough, and instead you'll want to pass a pre-filled struct to siphash. When doing this, it's important to always ensure the struct has no padding holes. The easiest way to do this is to simply arrange the members of the struct in descending order of size, and to use offsetofend() instead of sizeof() for getting the size. For performance reasons, if possible, it's probably a good thing to align the struct to the right boundary. Here's an example:

```
const struct {
        struct in6_addr saddr;
        u32 counter;
        u16 dport;
} __aligned(SIPHASH_ALIGNMENT) combined = {
```

```
        .saddr = *(struct in6_addr *)saddr,
        .counter = counter,
        .dport = dport
};
u64 h = siphash(&combined, offsetofend(typeof(combined), dport), &secret);
```

10.5 Resources

Read the SipHash paper if you're interested in learning more: https://131002.net/siphash/siphash.pdf

CHAPTER

ELEVEN

HALFSIPHASH - SIPHASH'S INSECURE YOUNGER COUSIN

Author

Written by Jason A. Donenfeld <jason@zx2c4.com>

On the off-chance that SipHash is not fast enough for your needs, you might be able to justify using HalfSipHash, a terrifying but potentially useful possibility. HalfSipHash cuts SipHash's rounds down from ”2-4” to ”1-3” and, even scarier, uses an easily brute-forcable 64-bit key (with a 32-bit output) instead of SipHash's 128-bit key. However, this may appeal to some high-performance *jhash* users.

HalfSipHash support is provided through the ”hsiphash” family of functions.

Warning: Do not ever use the hsiphash functions except for as a hashtable key function, and only then when you can be absolutely certain that the outputs will never be transmitted out of the kernel. This is only remotely useful over *jhash* as a means of mitigating hashtable flooding denial of service attacks.

On 64-bit kernels, the hsiphash functions actually implement SipHash-1-3, a reduced-round variant of SipHash, instead of HalfSipHash-1-3. This is because in 64-bit code, SipHash-1-3 is no slower than HalfSipHash-1-3, and can be faster. Note, this does *not* mean that in 64-bit kernels the hsiphash functions are the same as the siphash ones, or that they are secure; the hsiphash functions still use a less secure reduced-round algorithm and truncate their outputs to 32 bits.

11.1 Generating a hsiphash key

Keys should always be generated from a cryptographically secure source of random numbers, either using get_random_bytes or get_random_once:

```
hsiphash_key_t key;
get_random_bytes(&key, sizeof(key));
```

If you're not deriving your key from here, you're doing it wrong.

11.2 Using the hsiphash functions

There are two variants of the function, one that takes a list of integers, and one that takes a buffer:

```
u32 hsiphash(const void *data, size_t len, const hsiphash_key_t *key);
```

And:

```
u32 hsiphash_1u32(u32, const hsiphash_key_t *key);
u32 hsiphash_2u32(u32, u32, const hsiphash_key_t *key);
u32 hsiphash_3u32(u32, u32, u32, const hsiphash_key_t *key);
u32 hsiphash_4u32(u32, u32, u32, u32, const hsiphash_key_t *key);
```

If you pass the generic hsiphash function something of a constant length, it will constant fold at compile-time and automatically choose one of the optimized functions.

11.3 Hashtable key function usage

```
struct some_hashtable {
        DECLARE_HASHTABLE(hashtable, 8);
        hsiphash_key_t key;
};

void init_hashtable(struct some_hashtable *table)
{
        get_random_bytes(&table->key, sizeof(table->key));
}

static inline hlist_head *some_hashtable_bucket(struct some_hashtable *table,␣
↪struct interesting_input *input)
{
        return &table->hashtable[hsiphash(input, sizeof(*input), &table->key) &
↪ (HASH_SIZE(table->hashtable) - 1)];
}
```

You may then iterate like usual over the returned hash bucket.

11.4 Performance

hsiphash() is roughly 3 times slower than jhash(). For many replacements, this will not be a problem, as the hashtable lookup isn't the bottleneck. And in general, this is probably a good sacrifice to make for the security and DoS resistance of hsiphash().

CHAPTER

TWELVE

TRUSTED PLATFORM MODULE DOCUMENTATION

12.1 TPM Event Log

This document briefly describes what TPM log is and how it is handed over from the preboot firmware to the operating system.

12.1.1 Introduction

The preboot firmware maintains an event log that gets new entries every time something gets hashed by it to any of the PCR registers. The events are segregated by their type and contain the value of the hashed PCR register. Typically, the preboot firmware will hash the components to who execution is to be handed over or actions relevant to the boot process.

The main application for this is remote attestation and the reason why it is useful is nicely put in the very first section of [1]:

”Attestation is used to provide information about the platform’s state to a challenger. However, PCR contents are difficult to interpret; therefore, attestation is typically more useful when the PCR contents are accompanied by a measurement log. While not trusted on their own, the measurement log contains a richer set of information than do the PCR contents. The PCR contents are used to provide the validation of the measurement log.”

12.1.2 UEFI event log

UEFI provided event log has a few somewhat weird quirks.

Before calling ExitBootServices() Linux EFI stub copies the event log to a custom configuration table defined by the stub itself. Unfortunately, the events generated by ExitBootServices() don't end up in the table.

The firmware provides so called final events configuration table to sort out this issue. Events gets mirrored to this table after the first time EFI_TCG2_PROTOCOL.GetEventLog() gets called.

This introduces another problem: nothing guarantees that it is not called before the Linux EFI stub gets to run. Thus, it needs to calculate and save the final events table size while the stub is still running to the custom configuration table so that the TPM driver can later on skip these events when concatenating two halves of the event log from the custom configuration table and the final events table.

12.1.3 References

- [1] https://trustedcomputinggroup.org/resource/pc-client-specific-platform-firmware-profile-spe
- [2] The final concatenation is done in drivers/char/tpm/eventlog/efi.c

12.2 Virtual TPM Proxy Driver for Linux Containers

Authors:
Stefan Berger <stefanb@linux.vnet.ibm.com>

This document describes the virtual Trusted Platform Module (vTPM) proxy device driver for Linux containers.

12.2.1 Introduction

The goal of this work is to provide TPM functionality to each Linux container. This allows programs to interact with a TPM in a container the same way they interact with a TPM on the physical system. Each container gets its own unique, emulated, software TPM.

12.2.2 Design

To make an emulated software TPM available to each container, the container management stack needs to create a device pair consisting of a client TPM character device `/dev/tpmX` (with X=0,1,2...) and a 'server side' file descriptor. The former is moved into the container by creating a character device with the appropriate major and minor numbers while the file descriptor is passed to the TPM emulator. Software inside the container can then send TPM commands using the character device and the emulator will receive the commands via the file descriptor and use it for sending back responses.

To support this, the virtual TPM proxy driver provides a device `/dev/vtpmx` that is used to create device pairs using an ioctl. The ioctl takes as an input flags for configuring the device. The flags for example indicate whether TPM 1.2 or TPM 2 functionality is supported by the TPM emulator. The result of the ioctl are the file descriptor for the 'server side' as well as the major and minor numbers of the character device that was created. Besides that the number of the TPM character device is returned. If for example `/dev/tpm10` was created, the number (`dev_num`) 10 is returned.

Once the device has been created, the driver will immediately try to talk to the TPM. All commands from the driver can be read from the file descriptor returned by the ioctl. The commands should be responded to immediately.

12.2.3 UAPI

enum **vtpm_proxy_flags**
flags for the proxy TPM

Constants

VTPM_PROXY_FLAG_TPM2
the proxy TPM uses TPM 2.0 protocol

struct **vtpm_proxy_new_dev**
parameter structure for the VTPM_PROXY_IOC_NEW_DEV ioctl

Definition:

```
struct vtpm_proxy_new_dev {
    __u32 flags;
    __u32 tpm_num;
    __u32 fd;
    __u32 major;
    __u32 minor;
};
```

Members

flags
flags for the proxy TPM

tpm_num
index of the TPM device

fd
the file descriptor used by the proxy TPM

major
the major number of the TPM device

minor
the minor number of the TPM device

long **vtpmx_ioc_new_dev**(struct *file* *file, unsigned int ioctl, unsigned long arg)
handler for the VTPM_PROXY_IOC_NEW_DEV ioctl

Parameters

struct file *file
/dev/vtpmx

unsigned int ioctl
the ioctl number

unsigned long arg
pointer to the struct vtpmx_proxy_new_dev

Description

Creates an anonymous file that is used by the process acting as a TPM to communicate with the client processes. The function will also add a new TPM device through which data is proxied

to this TPM acting process. The caller will be provided with a file descriptor to communicate with the clients and major and minor numbers for the TPM device.

12.3 Virtual TPM interface for Xen

Authors: Matthew Fioravante (JHUAPL), Daniel De Graaf (NSA)

This document describes the virtual Trusted Platform Module (vTPM) subsystem for Xen. The reader is assumed to have familiarity with building and installing Xen, Linux, and a basic understanding of the TPM and vTPM concepts.

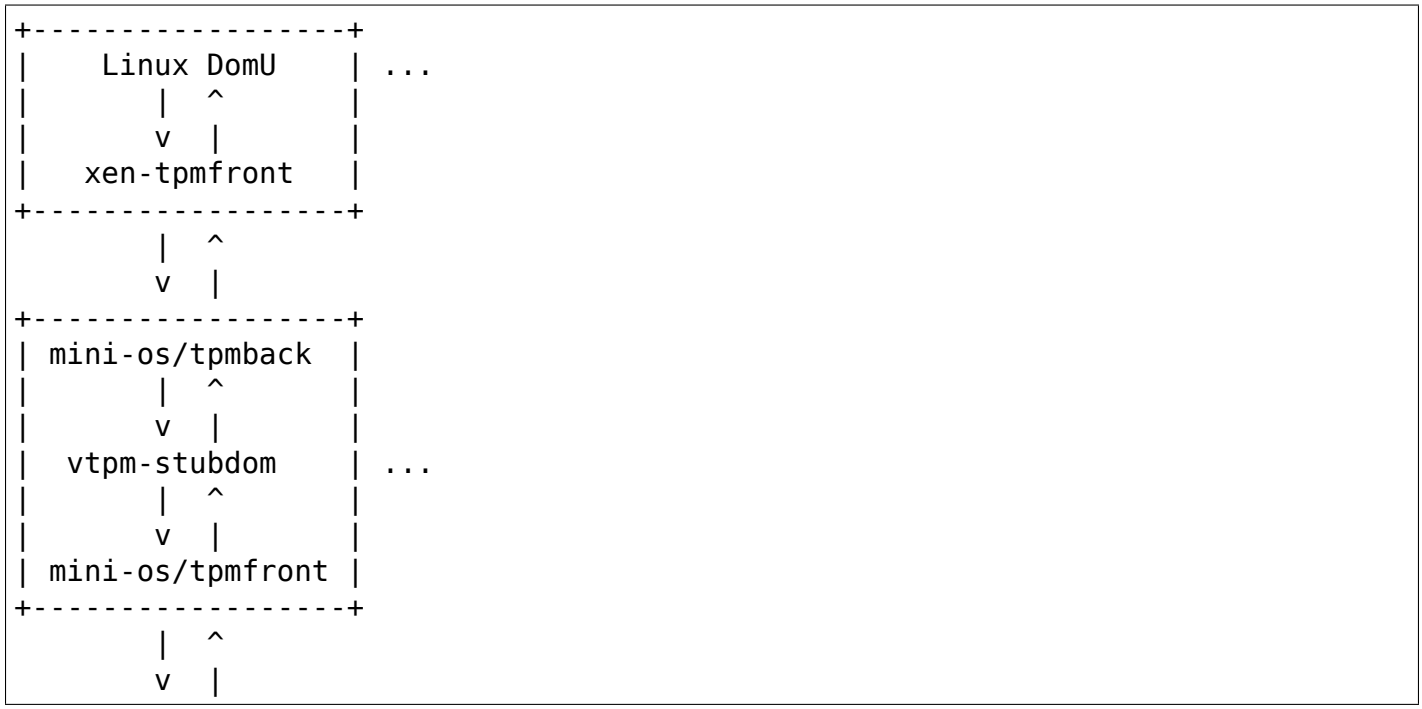
12.3.1 Introduction

The goal of this work is to provide a TPM functionality to a virtual guest operating system (in Xen terms, a DomU). This allows programs to interact with a TPM in a virtual system the same way they interact with a TPM on the physical system. Each guest gets its own unique, emulated, software TPM. However, each of the vTPM's secrets (Keys, NVRAM, etc) are managed by a vTPM Manager domain, which seals the secrets to the Physical TPM. If the process of creating each of these domains (manager, vTPM, and guest) is trusted, the vTPM subsystem extends the chain of trust rooted in the hardware TPM to virtual machines in Xen. Each major component of vTPM is implemented as a separate domain, providing secure separation guaranteed by the hypervisor. The vTPM domains are implemented in mini-os to reduce memory and processor overhead.

This mini-os vTPM subsystem was built on top of the previous vTPM work done by IBM and Intel corporation.

12.3.2 Design Overview

The architecture of vTPM is described below:



- **Linux DomU:**
  The Linux based guest that wants to use a vTPM. There may be more than one of these.
- **xen-tpmfront.ko:**
  Linux kernel virtual TPM frontend driver. This driver provides vTPM access to a Linux-based DomU.
- **mini-os/tpmback:**
  Mini-os TPM backend driver. The Linux frontend driver connects to this backend driver to facilitate communications between the Linux DomU and its vTPM. This driver is also used by vtpmmgr-stubdom to communicate with vtpm-stubdom.
- **vtpm-stubdom:**
  A mini-os stub domain that implements a vTPM. There is a one to one mapping between running vtpm-stubdom instances and logical vtpms on the system. The vTPM Platform Configuration Registers (PCRs) are normally all initialized to zero.
- **mini-os/tpmfront:**
  Mini-os TPM frontend driver. The vTPM mini-os domain vtpm-stubdom uses this driver to communicate with vtpmmgr-stubdom. This driver is also used in mini-os domains such as pv-grub that talk to the vTPM domain.
- **vtpmmgr-stubdom:**
  A mini-os domain that implements the vTPM manager. There is only one vTPM manager and it should be running during the entire lifetime of the machine. This domain regulates access to the physical TPM on the system and secures the persistent state of each vTPM.
- **mini-os/tpm_tis:**
  Mini-os TPM version 1.2 TPM Interface Specification (TIS) driver. This driver used by vtpmmgr-stubdom to talk directly to the hardware TPM. Communication is facilitated by mapping hardware memory pages into vtpmmgr-stubdom.
- **Hardware TPM:**
  The physical TPM that is soldered onto the motherboard.

12.3.3 Integration With Xen

Support for the vTPM driver was added in Xen using the libxl toolstack in Xen 4.3. See the Xen documentation (docs/misc/vtpm.txt) for details on setting up the vTPM and vTPM Manager stub domains. Once the stub domains are running, a vTPM device is set up in the same manner as a disk or network device in the domain's configuration file.

In order to use features such as IMA that require a TPM to be loaded prior to the initrd, the xen-tpmfront driver must be compiled in to the kernel. If not using such features, the driver can be compiled as a module and will be loaded as usual.

12.4 Firmware TPM Driver

This document describes the firmware Trusted Platform Module (fTPM) device driver.

12.4.1 Introduction

This driver is a shim for firmware implemented in ARM's TrustZone environment. The driver allows programs to interact with the TPM in the same way they would interact with a hardware TPM.

12.4.2 Design

The driver acts as a thin layer that passes commands to and from a TPM implemented in firmware. The driver itself doesn't contain much logic and is used more like a dumb pipe between firmware and kernel/userspace.

The firmware itself is based on the following paper: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/06/ftpm1.pdf

When the driver is loaded it will expose `/dev/tpmX` character devices to userspace which will enable userspace to communicate with the firmware TPM through this device.

CHAPTER

THIRTEEN

DIGITAL SIGNATURE VERIFICATION API

Author
Dmitry Kasatkin

Date
06.10.2011

13.1 Introduction

Digital signature verification API provides a method to verify digital signature. Currently digital signatures are used by the IMA/EVM integrity protection subsystem.

Digital signature verification is implemented using cut-down kernel port of GnuPG multi-precision integers (MPI) library. The kernel port provides memory allocation errors handling, has been refactored according to kernel coding style, and checkpatch.pl reported errors and warnings have been fixed.

Public key and signature consist of header and MPIs:

```
struct pubkey_hdr {
        uint8_t         version;        /* key format version */
        time_t          timestamp;      /* key made, always 0 for now */
        uint8_t         algo;
        uint8_t         nmpi;
        char            mpi[0];
} __packed;

struct signature_hdr {
        uint8_t         version;        /* signature format version */
        time_t          timestamp;      /* signature made */
        uint8_t         algo;
        uint8_t         hash;
        uint8_t         keyid[8];
        uint8_t         nmpi;
        char            mpi[0];
} __packed;
```

keyid equals to SHA1[12-19] over the total key content. Signature header is used as an input to generate a signature. Such approach insures that key or signature header could not be changed. It protects timestamp from been changed and can be used for rollback protection.

13.2 API

API currently includes only 1 function:

```
digsig_verify() - digital signature verification with public key


/**
* digsig_verify() - digital signature verification with public key
* @keyring:     keyring to search key in
* @sig: digital signature
* @sigen:       length of the signature
* @data:        data
* @datalen:     length of the data
* @return:      0 on success, -EINVAL otherwise
*
* Verifies data integrity against digital signature.
* Currently only RSA is supported.
* Normally hash of the content is used as a data for this function.
*
*/
int digsig_verify(struct key *keyring, const char *sig, int siglen,
                  const char *data, int datalen);
```

13.3 User-space utilities

The signing and key management utilities evm-utils provide functionality to generate signatures, to load keys into the kernel keyring. Keys can be in PEM or converted to the kernel format. When the key is added to the kernel keyring, the keyid defines the name of the key: 5D2B05FC633EE3E8 in the example below.

Here is example output of the keyctl utility:

```
$ keyctl show
Session Keyring
-3 --alswrv      0     0  keyring: _ses
603976250 --alswrv      0    -1   \_ keyring: _uid.0
817777377 --alswrv      0     0       \_ user: kmk
891974900 --alswrv      0     0       \_ encrypted: evm-key
170323636 --alswrv      0     0       \_ keyring: _module
548221616 --alswrv      0     0       \_ keyring: _ima
128198054 --alswrv      0     0       \_ keyring: _evm

$ keyctl list 128198054
1 key in keyring:
620789745: --alswrv     0     0 user: 5D2B05FC633EE3E8
```

CHAPTER

FOURTEEN

LANDLOCK LSM: KERNEL DOCUMENTATION

Author
Mickaël Salaün

Date
December 2022

Landlock's goal is to create scoped access-control (i.e. sandboxing). To harden a whole system, this feature should be available to any process, including unprivileged ones. Because such process may be compromised or backdoored (i.e. untrusted), Landlock's features must be safe to use from the kernel and other processes point of view. Landlock's interface must therefore expose a minimal attack surface.

Landlock is designed to be usable by unprivileged processes while following the system security policy enforced by other access control mechanisms (e.g. DAC, LSM). Indeed, a Landlock rule shall not interfere with other access-controls enforced on the system, only add more restrictions.

Any user can enforce Landlock rulesets on their processes. They are merged and evaluated according to the inherited ones in a way that ensures that only more constraints can be added.

User space documentation can be found here: Documentation/userspace-api/landlock.rst.

14.1 Guiding principles for safe access controls

- A Landlock rule shall be focused on access control on kernel objects instead of syscall filtering (i.e. syscall arguments), which is the purpose of seccomp-bpf.
- To avoid multiple kinds of side-channel attacks (e.g. leak of security policies, CPU-based attacks), Landlock rules shall not be able to programmatically communicate with user space.
- Kernel access check shall not slow down access request from unsandboxed processes.
- Computation related to Landlock operations (e.g. enforcing a ruleset) shall only impact the processes requesting them.
- Resources (e.g. file descriptors) directly obtained from the kernel by a sandboxed process shall retain their scoped accesses (at the time of resource acquisition) whatever process use them. Cf. *File descriptor access rights*.

14.2 Design choices

14.2.1 Inode access rights

All access rights are tied to an inode and what can be accessed through it. Reading the content of a directory does not imply to be allowed to read the content of a listed inode. Indeed, a file name is local to its parent directory, and an inode can be referenced by multiple file names thanks to (hard) links. Being able to unlink a file only has a direct impact on the directory, not the unlinked inode. This is the reason why `LANDLOCK_ACCESS_FS_REMOVE_FILE` or `LANDLOCK_ACCESS_FS_REFER` are not allowed to be tied to files but only to directories.

14.2.2 File descriptor access rights

Access rights are checked and tied to file descriptors at open time. The underlying principle is that equivalent sequences of operations should lead to the same results, when they are executed under the same Landlock domain.

Taking the `LANDLOCK_ACCESS_FS_TRUNCATE` right as an example, it may be allowed to open a file for writing without being allowed to *`ftruncate`* the resulting file descriptor if the related file hierarchy doesn't grant such access right. The following sequences of operations have the same semantic and should then have the same result:

- `truncate(path);`
- `int fd = open(path, O_WRONLY); ftruncate(fd); close(fd);`

Similarly to file access modes (e.g. `O_RDWR`), Landlock access rights attached to file descriptors are retained even if they are passed between processes (e.g. through a Unix domain socket). Such access rights will then be enforced even if the receiving process is not sandboxed by Landlock. Indeed, this is required to keep a consistent access control over the whole system, and this avoids unattended bypasses through file descriptor passing (i.e. confused deputy attack).

14.3 Tests

Userspace tests for backward compatibility, ptrace restrictions and filesystem support can be found here: tools/testing/selftests/landlock/.

14.4 Kernel structures

14.4.1 Object

struct **`landlock_object_underops`**
Operations on an underlying object

Definition:

```
struct landlock_object_underops {
    void (*release)(struct landlock_object *const object) __releases(object->
↪lock);
};
```

Members

release
Releases the underlying object (e.g. iput() for an inode).

struct **landlock_object**
Security blob tied to a kernel object

Definition:

```
struct landlock_object {
    refcount_t usage;
    spinlock_t lock;
    void *underobj;
    union {
        struct rcu_head rcu_free;
        const struct landlock_object_underops *underops;
    };
};
```

Members

usage
This counter is used to tie an object to the rules matching it or to keep it alive while adding a new rule. If this counter reaches zero, this struct must not be modified, but this counter can still be read from within an RCU read-side critical section. When adding a new rule to an object with a usage counter of zero, we must wait until the pointer to this object is set to NULL (or recycled).

lock
Protects against concurrent modifications. This lock must be held from the time **usage** drops to zero until any weak references from **underobj** to this object have been cleaned up.

Lock ordering: inode->i_lock nests inside this.

underobj
Used when cleaning up an object and to mark an object as tied to its underlying kernel structure. This pointer is protected by **lock**. Cf. landlock_release_inodes() and release_inode().

{unnamed_union}
anonymous

rcu_free
Enables lockless use of **usage**, **lock** and **underobj** from within an RCU read-side critical section. **rcu_free** and **underops** are only used by landlock_put_object().

underops
Enables landlock_put_object() to release the underlying object (e.g. inode).

Description

The goal of this structure is to enable to tie a set of ephemeral access rights (pertaining to different domains) to a kernel object (e.g an inode) in a safe way. This implies to handle concurrent use and modification.

The lifetime of a *struct landlock_object* depends on the rules referring to it.

14.4.2 Filesystem

struct **landlock_inode_security**
Inode security blob

Definition:

```
struct landlock_inode_security {
    struct landlock_object __rcu *object;
};
```

Members

object
Weak pointer to an allocated object. All assignments of a new object are protected by the underlying inode->i_lock. However, atomically disassociating **object** from the inode is only protected by **object->lock**, from the time **object**'s usage refcount drops to zero to the time this pointer is nulled out (cf. release_inode() and hook_sb_delete()). Indeed, such disassociation doesn't require inode->i_lock thanks to the careful rcu_access_pointer() check performed by get_inode_object().

Description

Enable to reference a *struct landlock_object* tied to an inode (i.e. underlying object).

struct **landlock_file_security**
File security blob

Definition:

```
struct landlock_file_security {
    access_mask_t allowed_access;
};
```

Members

allowed_access
Access rights that were available at the time of opening the file. This is not necessarily the full set of access rights available at that time, but it's the necessary subset as needed to authorize later operations on the open file.

Description

This information is populated when opening a file in hook_file_open, and tracks the relevant Landlock access rights that were available at the time of opening the file. Other LSM hooks use these rights in order to authorize operations on already opened files.

struct **landlock_superblock_security**

Superblock security blob

Definition:

```
struct landlock_superblock_security {
    atomic_long_t inode_refs;
};
```

Members

inode_refs

Number of pending inodes (from this superblock) that are being released by release_inode(). Cf. struct super_block->s_fsnotify_inode_refs .

Description

Enable hook_sb_delete() to wait for concurrent calls to release_inode().

14.4.3 Ruleset and domain

A domain is a read-only ruleset tied to a set of subjects (i.e. tasks' credentials). Each time a ruleset is enforced on a task, the current domain is duplicated and the ruleset is imported as a new layer of rules in the new domain. Indeed, once in a domain, each rule is tied to a layer level. To grant access to an object, at least one rule of each layer must allow the requested action on the object. A task can then only transit to a new domain that is the intersection of the constraints from the current domain and those of a ruleset provided by the task.

The definition of a subject is implicit for a task sandboxing itself, which makes the reasoning much easier and helps avoid pitfalls.

struct **landlock_layer**

Access rights for a given layer

Definition:

```
struct landlock_layer {
    u16 level;
    access_mask_t access;
};
```

Members

level

Position of this layer in the layer stack.

access

Bitfield of allowed actions on the kernel object. They are relative to the object type (e.g. `LANDLOCK_ACTION_FS_READ`).

union **landlock_key**

Key of a ruleset's red-black tree

Definition:

```
union landlock_key {
    struct landlock_object *object;
    uintptr_t data;
};
```

Members

object
Pointer to identify a kernel object (e.g. an inode).

data
Raw data to identify an arbitrary 32-bit value (e.g. a TCP port).

enum **landlock_key_type**
Type of *union landlock_key*

Constants

LANDLOCK_KEY_INODE
Type of *landlock_ruleset.root_inode*'s node keys.

LANDLOCK_KEY_NET_PORT
Type of *landlock_ruleset.root_net_port*'s node keys.

struct **landlock_id**
Unique rule identifier for a ruleset

Definition:

```
struct landlock_id {
    union landlock_key key;
    const enum landlock_key_type type;
};
```

Members

key
Identifies either a kernel object (e.g. an inode) or a raw value (e.g. a TCP port).

type
Type of a landlock_ruleset's root tree.

struct **landlock_rule**
Access rights tied to an object

Definition:

```
struct landlock_rule {
    struct rb_node node;
    union landlock_key key;
    u32 num_layers;
    struct landlock_layer layers[] ;
};
```

Members

node
Node in the ruleset's red-black tree.

key
A union to identify either a kernel object (e.g. an inode) or a raw data value (e.g. a network socket port). This is used as a key for this ruleset element. The pointer is set once and never modified. It always points to an allocated object because each rule increments the refcount of its object.

num_layers
Number of entries in **layers**.

layers
Stack of layers, from the latest to the newest, implemented as a flexible array member (FAM).

struct **landlock_hierarchy**
Node in a ruleset hierarchy

Definition:

```
struct landlock_hierarchy {
    struct landlock_hierarchy *parent;
    refcount_t usage;
};
```

Members

parent
Pointer to the parent node, or NULL if it is a root Landlock domain.

usage
Number of potential children domains plus their parent domain.

struct **landlock_ruleset**
Landlock ruleset

Definition:

```
struct landlock_ruleset {
    struct rb_root root_inode;
#if IS_ENABLED(CONFIG_INET);
    struct rb_root root_net_port;
#endif ;
    struct landlock_hierarchy *hierarchy;
    union {
        struct work_struct work_free;
        struct {
            struct mutex lock;
            refcount_t usage;
            u32 num_rules;
            u32 num_layers;
            access_masks_t access_masks[];
        };
    };
};
```

Members

root_inode
Root of a red-black tree containing *struct landlock_rule* nodes with inode object. Once a ruleset is tied to a process (i.e. as a domain), this tree is immutable until **usage** reaches zero.

root_net_port
Root of a red-black tree containing *struct landlock_rule* nodes with network port. Once a ruleset is tied to a process (i.e. as a domain), this tree is immutable until **usage** reaches zero.

hierarchy
Enables hierarchy identification even when a parent domain vanishes. This is needed for the ptrace protection.

{unnamed_union}
anonymous

work_free
Enables to free a ruleset within a lockless section. This is only used by landlock_put_ruleset_deferred() when **usage** reaches zero. The fields **lock**, **usage**, **num_rules**, **num_layers** and **access_masks** are then unused.

{unnamed_struct}
anonymous

lock
Protects against concurrent modifications of **root**, if **usage** is greater than zero.

usage
Number of processes (i.e. domains) or file descriptors referencing this ruleset.

num_rules
Number of non-overlapping (i.e. not for the same object) rules in this ruleset.

num_layers
Number of layers that are used in this ruleset. This enables to check that all the layers allow an access request. A value of 0 identifies a non-merged ruleset (i.e. not a domain).

access_masks
Contains the subset of filesystem and network actions that are restricted by a ruleset. A domain saves all layers of merged rulesets in a stack (FAM), starting from the first layer to the last one. These layers are used when merging rulesets, for user space backward compatibility (i.e. future-proof), and to properly handle merged rulesets without overlapping access rights. These layers are set once and never changed for the lifetime of the ruleset.

Description

This data structure must contain unique entries, be updatable, and quick to match an object.

CHAPTER

FIFTEEN

SECRETS DOCUMENTATION

15.1 Confidential Computing secrets

This document describes how Confidential Computing secret injection is handled from the firmware to the operating system, in the EFI driver and the efi_secret kernel module.

15.1.1 Introduction

Confidential Computing (coco) hardware such as AMD SEV (Secure Encrypted Virtualization) allows guest owners to inject secrets into the VMs memory without the host/hypervisor being able to read them. In SEV, secret injection is performed early in the VM launch process, before the guest starts running.

The efi_secret kernel module allows userspace applications to access these secrets via securityfs.

15.1.2 Secret data flow

The guest firmware may reserve a designated memory area for secret injection, and publish its location (base GPA and length) in the EFI configuration table under a `LINUX_EFI_COCO_SECRET_AREA_GUID` entry (`adf956ad-e98c-484c-ae11-b51c7d336447`). This memory area should be marked by the firmware as `EFI_RESERVED_TYPE`, and therefore the kernel should not be use it for its own purposes.

During the VM's launch, the virtual machine manager may inject a secret to that area. In AMD SEV and SEV-ES this is performed using the `KVM_SEV_LAUNCH_SECRET` command (see [sev]). The structure of the injected Guest Owner secret data should be a GUIDed table of secret values; the binary format is described in `drivers/virt/coco/efi_secret/efi_secret.c` under ”Structure of the EFI secret area”.

On kernel start, the kernel's EFI driver saves the location of the secret area (taken from the EFI configuration table) in the `efi.coco_secret` field. Later it checks if the secret area is populated: it maps the area and checks whether its content begins with `EFI_SECRET_TABLE_HEADER_GUID` (`1e74f542-71dd-4d66-963e-ef4287ff173b`). If the secret area is populated, the EFI driver will autoload the efi_secret kernel module, which exposes the secrets to userspace applications via securityfs. The details of the efi_secret filesystem interface are in [secrets-coco-abi].

15.1.3 Application usage example

Consider a guest performing computations on encrypted files. The Guest Owner provides the decryption key (= secret) using the secret injection mechanism. The guest application reads the secret from the efi_secret filesystem and proceeds to decrypt the files into memory and then performs the needed computations on the content.

In this example, the host can't read the files from the disk image because they are encrypted. Host can't read the decryption key because it is passed using the secret injection mechanism (= secure channel). Host can't read the decrypted content from memory because it's a confidential (memory-encrypted) guest.

Here is a simple example for usage of the efi_secret module in a guest to which an EFI secret area with 4 secrets was injected during launch:

```
# ls -la /sys/kernel/security/secrets/coco
total 0
drwxr-xr-x 2 root root 0 Jun 28 11:54 .
drwxr-xr-x 3 root root 0 Jun 28 11:54 ..
-r--r----- 1 root root 0 Jun 28 11:54 736870e5-84f0-4973-92ec-06879ce3da0b
-r--r----- 1 root root 0 Jun 28 11:54 83c83f7f-1356-4975-8b7e-d3a0b54312c6
-r--r----- 1 root root 0 Jun 28 11:54 9553f55d-3da2-43ee-ab5d-ff17f78864d2
-r--r----- 1 root root 0 Jun 28 11:54 e6f5a162-d67f-4750-a67c-5d065f2a9910

# hd /sys/kernel/security/secrets/coco/e6f5a162-d67f-4750-a67c-5d065f2a9910
00000000  74 68 65 73 65 2d 61 72  65 2d 74 68 65 2d 6b 61  |these-are-the-ka|
00000010  74 61 2d 73 65 63 72 65  74 73 00 01 02 03 04 05  |ta-secrets......|
00000020  06 07                                             |..|
00000022

# rm /sys/kernel/security/secrets/coco/e6f5a162-d67f-4750-a67c-5d065f2a9910

# ls -la /sys/kernel/security/secrets/coco
total 0
drwxr-xr-x 2 root root 0 Jun 28 11:55 .
drwxr-xr-x 3 root root 0 Jun 28 11:54 ..
-r--r----- 1 root root 0 Jun 28 11:54 736870e5-84f0-4973-92ec-06879ce3da0b
-r--r----- 1 root root 0 Jun 28 11:54 83c83f7f-1356-4975-8b7e-d3a0b54312c6
-r--r----- 1 root root 0 Jun 28 11:54 9553f55d-3da2-43ee-ab5d-ff17f78864d2
```

15.1.4 References

See [sev-api-spec] for more info regarding SEV `LAUNCH_SECRET` operation.

BIBLIOGRAPHY

[sev] Documentation/virt/kvm/x86/amd-memory-encryption.rst

[secrets-coco-abi] Documentation/ABI/testing/securityfs-secrets-coco

[sev-api-spec] https://www.amd.com/system/files/TechDocs/55766_SEV-KM_API_Specification.pdf

INDEX

V