CREDENTIALS IN LINUX

By: David Howells <dhowells@redhat.com>

Contents:

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.

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/keys.txt.

(5) LSM

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

alternate LSM options: SELinux and Smack.

Both 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.

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.

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.

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) Whilst 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.

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
	void *current_security(void)	Current's LSM security pointer
	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.

ACCESSING ANOTHER TASK'S CREDENTIALS

Whilst 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 whilst 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.

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 whilst 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 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);
        }
```

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.

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).

```
======================================
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.