NAME

capabilities − overview of Linux capabilities

DESCRIPTION

For the purpose of performing permission checks, traditional Unix implementations distinguish two categories of processes: *privileged* processes (whose effective user ID is 0, referred to as superuser or root), and *unprivileged* processes (whose effective UID is non-zero). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).

Starting with kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as *capabilities*, which can be independently enabled and disabled. Capabilities are a per-thread attribute.

Capabilities List

The following list shows the capabilities implemented on Linux, and the operations or behaviors that each capability permits:

CAP_AUDIT_CONTROL (since Linux 2.6.11)
Enable and disable kernel auditing; change auditing filter rules; retrieve auditing status and filtering rules.

CAP_AUDIT_WRITE (since Linux 2.6.11)
Write records to kernel auditing log.

CAP_CHOWN
Make arbitrary changes to file UIDs and GIDs (see **chown**(2)).

CAP_DAC_OVERRIDE
Bypass file read, write, and execute permission checks. (DAC is an abbreviation of "discretionary access control".)

CAP_DAC_READ_SEARCH
Bypass file read permission checks and directory read and execute permission checks.

CAP_FOWNER
* Bypass permission checks on operations that normally require the file system UID of the process to match the UID of the file (e.g., **chmod**(2), **utime**(2)), excluding those operations covered by **CAP_DAC_OVERRIDE** and **CAP_DAC_READ_SEARCH**;
* set extended file attributes (see **chattr**(1)) on arbitrary files;
* set Access Control Lists (ACLs) on arbitrary files;
* ignore directory sticky bit on file deletion;
* specify **O_NOATIME** for arbitrary files in **open**(2) and **fcntl**(2).

CAP_FSETID
Don't clear set-user-ID and set-group-ID permission bits when a file is modified; set the set-group-ID bit for a file whose GID does not match the file system or any of the supplementary GIDs of the calling process.

CAP_IPC_LOCK
Lock memory (**mlock**(2), **mlockall**(2), **mmap**(2), **shmctl**(2)).

CAP_IPC_OWNER
Bypass permission checks for operations on System V IPC objects.

CAP_KILL
Bypass permission checks for sending signals (see **kill**(2)). This includes use of the **ioctl**(2) **KDSIGACCEPT** operation.

CAP_LEASE (since Linux 2.4)
Establish leases on arbitrary files (see **fcntl**(2)).

CAP_LINUX_IMMUTABLE
Set the **FS_APPEND_FL** and **FS_IMMUTABLE_FL** i-node flags (see **chattr**(1)).

CAP_MAC_ADMIN (since Linux 2.6.25)
Override Mandatory Access Control (MAC). Implemented for the Smack Linux Security Module (LSM).

CAP_MAC_OVERRIDE (since Linux 2.6.25)
Allow MAC configuration or state changes. Implemented for the Smack LSM.

CAP_MKNOD (since Linux 2.4)
Create special files using **mknod**(2).

CAP_NET_ADMIN
Perform various network-related operations (e.g., setting privileged socket options, enabling multicasting, interface configuration, modifying routing tables).

CAP_NET_BIND_SERVICE
Bind a socket to Internet domain privileged ports (port numbers less than 1024).

CAP_NET_BROADCAST
(Unused) Make socket broadcasts, and listen to multicasts.

CAP_NET_RAW
Use RAW and PACKET sockets.

CAP_SETGID
Make arbitrary manipulations of process GIDs and supplementary GID list; forge GID when passing socket credentials via Unix domain sockets.

CAP_SETFCAP (since Linux 2.6.24)
Set file capabilities.

CAP_SETPCAP
If file capabilities are not supported: grant or remove any capability in the caller's permitted capability set to or from any other process. (This property of **CAP_SETPCAP** is not available when the kernel is configured to support file capabilities, since **CAP_SETPCAP** has entirely different semantics for such kernels.)

If file capabilities are supported: add any capability from the calling thread's bounding set to its inheritable set; drop capabilities from the bounding set (via **prctl**(2) **PR_CAPBSET_DROP**); make changes to the *securebits* flags.

CAP_SETUID
Make arbitrary manipulations of process UIDs (**setuid**(2), **setreuid**(2), **setresuid**(2), **setfsuid**(2)); make forged UID when passing socket credentials via Unix domain sockets.

CAP_SYS_ADMIN
* Perform a range of system administration operations including: **quotactl**(2), **mount**(2), **umount**(2), **swapon**(2), **swapoff**(2), **sethostname**(2), and **setdomainname**(2);
* perform **IPC_SET** and **IPC_RMID** operations on arbitrary System V IPC objects;
* perform operations on *trusted* and *security* Extended Attributes (see **attr**(5));
* use **lookup_dcookie**(2);
* use **ioprio_set**(2) to assign **IOPRIO_CLASS_RT** and (before Linux 2.6.25) **IOPRIO_CLASS_IDLE** I/O scheduling classes;
* forge UID when passing socket credentials;
* exceed */proc/sys/fs/file-max*, the system-wide limit on the number of open files, in system calls that open files (e.g., **accept**(2), **execve**(2), **open**(2), **pipe**(2));
* employ **CLONE_NEWNS** flag with **clone**(2) and **unshare**(2);
* perform **KEYCTL_CHOWN** and **KEYCTL_SETPERM keyctl**(2) operations.

CAP_SYS_BOOT
Use **reboot**(2) and **kexec_load**(2).

CAP_SYS_CHROOT
Use **chroot**(2).

CAP_SYS_MODULE
Load and unload kernel modules (see **init_module**(2) and **delete_module**(2)); in kernels before 2.6.25: drop capabilities from the system-wide capability bounding set.

CAP_SYS_NICE
* Raise process nice value (**nice**(2), **setpriority**(2)) and change the nice value for arbitrary processes;
* set real-time scheduling policies for calling process, and set scheduling policies and priorities for arbitrary processes (**sched_setscheduler**(2), **sched_setparam**(2));
* set CPU affinity for arbitrary processes (**sched_setaffinity**(2));
* set I/O scheduling class and priority for arbitrary processes (**ioprio_set**(2));
* apply **migrate_pages**(2) to arbitrary processes and allow processes to be migrated to arbitrary nodes;
* apply **move_pages**(2) to arbitrary processes;
* use the **MPOL_MF_MOVE_ALL** flag with **mbind**(2) and **move_pages**(2).

CAP_SYS_PACCT
Use **acct**(2).

CAP_SYS_PTRACE
Trace arbitrary processes using **ptrace**(2)

CAP_SYS_RAWIO
Perform I/O port operations (**iopl**(2) and **ioperm**(2)); access */proc/kcore*.

CAP_SYS_RESOURCE
* Use reserved space on ext2 file systems;
* make **ioctl**(2) calls controlling ext3 journaling;
* override disk quota limits;
* increase resource limits (see **setrlimit**(2));
* override **RLIMIT_NPROC** resource limit;
* raise *msg_qbytes* limit for a System V message queue above the limit in */proc/sys/kernel/msgmnb* (see **msgop**(2) and **msgctl**(2)).

CAP_SYS_TIME
Set system clock (**settimeofday**(2), **stime**(2), **adjtimex**(2)); set real-time (hardware) clock.

CAP_SYS_TTY_CONFIG
Use **vhangup**(2).

Past and Current Implementation
A full implementation of capabilities requires that:

1. For all privileged operations, the kernel must check whether the thread has the required capability in its effective set.
2. The kernel must provide system calls allowing a thread's capability sets to be changed and retrieved.
3. The file system must support attaching capabilities to an executable file, so that a process gains those capabilities when the file is executed.

Before kernel 2.6.24, only the first two of these requirements are met; since kernel 2.6.24, all three requirements are met.

Thread Capability Sets
Each thread has three capability sets containing zero or more of the above capabilities:

Permitted:
This is a limiting superset for the effective capabilities that the thread may assume. It is also a limiting superset for the capabilities that may be added to the inheritable set by a thread that does not have the **CAP_SETPCAP** capability in its effective set.

If a thread drops a capability from its permitted set, it can never re-acquire that capability (unless it **execve**(2)s either a set-user-ID-root program, or a program whose associated file capabilities grant that capability).

Inheritable:
This is a set of capabilities preserved across an **execve**(2). It provides a mechanism for a process to assign capabilities to the permitted set of the new program during an **execve**(2).

Effective:
This is the set of capabilities used by the kernel to perform permission checks for the thread.

A child created via **fork**(2) inherits copies of its parent's capability sets. See below for a discussion of the treatment of capabilities during **execve**(2).

Using **capset**(2), a thread may manipulate its own capability sets (see below).

File Capabilities

Since kernel 2.6.24, the kernel supports associating capability sets with an executable file using **setcap**(8). The file capability sets are stored in an extended attribute (see **setxattr**(2)) named *security.capability*. Writing to this extended attribute requires the **CAP_SETFCAP** capability. The file capability sets, in conjunction with the capability sets of the thread, determine the capabilities of a thread after an **execve**(2).

The three file capability sets are:

Permitted (formerly known as *forced*):
These capabilities are automatically permitted to the thread, regardless of the thread's inheritable capabilities.

Inheritable (formerly known as *allowed*):
This set is ANDed with the thread's inheritable set to determine which inheritable capabilities are enabled in the permitted set of the thread after the **execve**(2).

Effective:
This is not a set, but rather just a single bit. If this bit is set, then during an **execve**(2) all of the new permitted capabilities for the thread are also raised in the effective set. If this bit is not set, then after an **execve**(2), none of the new permitted capabilities is in the new effective set.

Enabling the file effective capability bit implies that any file permitted or inheritable capability that causes a thread to acquire the corresponding permitted capability during an **execve**(2) (see the transformation rules described below) will also acquire that capability in its effective set. Therefore, when assigning capabilities to a file (**setcap**(8), **cap_set_file**(3), **cap_set_fd**(3)), if we specify the effective flag as being enabled for any capability, then the effective flag must also be specified as enabled for all other capabilities for which the corresponding permitted or inheritable flags is enabled.

Transformation of Capabilities During execve()

During an **execve**(2), the kernel calculates the new capabilities of the process using the following algorithm:

```
P'(permitted) = (P(inheritable) & F(inheritable)) |
                (F(permitted) & cap_bset)

P'(effective) = F(effective) ? P'(permitted) : 0

P'(inheritable) = P(inheritable)    [i.e., unchanged]
```

where:

P	denotes the value of a thread capability set before the **execve**(2)
P'	denotes the value of a capability set after the **execve**(2)
F	denotes a file capability set
cap_bset	is the value of the capability bounding set (described below).

Capabilities and execution of programs by root

In order to provide an all-powerful *root* using capability sets, during an **execve**(2):

1. If a set-user-ID-root program is being executed, or the real user ID of the process is 0 (root) then the file inheritable and permitted sets are defined to be all ones (i.e., all capabilities enabled).
2. If a set-user-ID-root program is being executed, then the file effective bit is defined to be one (enabled).

The upshot of the above rules, combined with the capabilities transformations described above, is that when a process **execve**(2)s a set-user-ID-root program, or when a process with an effective UID of 0 **execve**(2)s a program, it gains all capabilities in its permitted and effective capability sets, except those masked out by the capability bounding set. This provides semantics that are the same as those provided by traditional Unix systems.

Capability bounding set

The capability bounding set is a security mechanism that can be used to limit the capabilities that can be gained during an **execve**(2). The bounding set is used in the following ways:

* During an **execve**(2), the capability bounding set is ANDed with the file permitted capability set, and the result of this operation is assigned to the thread's permitted capability set. The capability bounding set thus places a limit on the permitted capabilities that may be granted by an executable file.
* (Since Linux 2.6.25) The capability bounding set acts as a limiting superset for the capabilities that a thread can add to its inheritable set using **capset**(2). This means that if the capability is not in the bounding set, then a thread can't add one of its permitted capabilities to its inheritable set and thereby have that capability preserved in its permitted set when it **execve**(2)s a file that has the capability in its inheritable set.

Note that the bounding set masks the file permitted capabilities, but not the inherited capabilities. If a thread maintains a capability in its inherited set that is not in its bounding set, then it can still gain that capability in its permitted set by executing a file that has the capability in its inherited set.

Depending on the kernel version, the capability bounding set is either a system-wide attribute, or a per-process attribute.

Capability bounding set prior to Linux 2.6.25

In kernels before 2.6.25, the capability bounding set is a system-wide attribute that affects all threads on the system. The bounding set is accessible via the file */proc/sys/kernel/cap-bound*. (Confusingly, this bit mask parameter is expressed as a signed decimal number in */proc/sys/kernel/cap-bound*.)

Only the **init** process may set capabilities in the capability bounding set; other than that, the superuser (more precisely: programs with the **CAP_SYS_MODULE** capability) may only clear capabilities from this set.

On a standard system the capability bounding set always masks out the **CAP_SETPCAP** capability. To remove this restriction (dangerous!), modify the definition of **CAP_INIT_EFF_SET** in *include/linux/capability.h* and rebuild the kernel.

The system-wide capability bounding set feature was added to Linux starting with kernel version 2.2.11.

Capability bounding set from Linux 2.6.25 onwards

From Linux 2.6.25, the *capability bounding set* is a per-thread attribute. (There is no longer a system-wide capability bounding set.)

The bounding set is inherited at **fork**(2) from the thread's parent, and is preserved across an **execve**(2).

A thread may remove capabilities from its capability bounding set using the **prctl**(2) **PR_CAPBSET_DROP** operation, provided it has the **CAP_SETPCAP** capability. Once a capability has been dropped from the bounding set, it cannot be restored to that set. A thread can determine if a capability is in its bounding set using the **prctl**(2) **PR_CAPBSET_READ** operation.

Removing capabilities from the bounding set is only supported if file capabilities are compiled into the kernel (CONFIG_SECURITY_FILE_CAPABILITIES). In that case, the **init** process (the ancestor of all processes) begins with a full bounding set. If file capabilities are not compiled into the kernel, then **init** begins with a full bounding set minus **CAP_SETPCAP**, because this capability has a different meaning when there are no file capabilities.

Removing a capability from the bounding set does not remove it from the thread's inherited set. However it does prevent the capability from being added back into the thread's inherited set in the future.

Effect of User ID Changes on Capabilities

To preserve the traditional semantics for transitions between 0 and non-zero user IDs, the kernel makes the following changes to a thread's capability sets on changes to the thread's real, effective, saved set, and file system user IDs (using **setuid**(2), **setresuid**(2), or similar):

1. If one or more of the real, effective or saved set user IDs was previously 0, and as a result of the UID changes all of these IDs have a non-zero value, then all capabilities are cleared from the permitted and effective capability sets.
2. If the effective user ID is changed from 0 to non-zero, then all capabilities are cleared from the effective set.
3. If the effective user ID is changed from non-zero to 0, then the permitted set is copied to the effective set.
4. If the file system user ID is changed from 0 to non-zero (see **setfsuid**(2)) then the following capabilities are cleared from the effective set: **CAP_CHOWN**, **CAP_DAC_OVERRIDE**, **CAP_DAC_READ_SEARCH**, **CAP_FOWNER**, **CAP_FSETID**, and **CAP_MAC_OVERRIDE**. If the file system UID is changed from non-zero to 0, then any of these capabilities that are enabled in the permitted set are enabled in the effective set.

If a thread that has a 0 value for one or more of its user IDs wants to prevent its permitted capability set being cleared when it resets all of its user IDs to non-zero values, it can do so using the **prctl**(2) **PR_SET_KEEPCAPS** operation.

Programmatically adjusting capability sets

A thread can retrieve and change its capability sets using the **capget**(2) and **capset**(2) system calls. However, the use of **cap_get_proc**(3) and **cap_set_proc**(3), both provided in the *libcap* package, is preferred for this purpose. The following rules govern changes to the thread capability sets:

1. If the caller does not have the **CAP_SETPCAP** capability, the new inheritable set must be a subset of the combination of the existing inheritable and permitted sets.
2. (Since kernel 2.6.25) The new inheritable set must be a subset of the combination of the existing inheritable set and the capability bounding set.
3. The new permitted set must be a subset of the existing permitted set (i.e., it is not possible to acquire permitted capabilities that the thread does not currently have).
4. The new effective set must be a subset of the new permitted set.

The "securebits" flags: establishing a capabilities-only environment

Starting with kernel 2.6.26, and with a kernel in which file capabilities are enabled, Linux implements a set of per-thread *securebits* flags that can be used to disable special handling of capabilities for UID 0 (*root*). These flags are as follows:

SECURE_KEEP_CAPS
Setting this flag allows a thread that has one or more 0 UIDs to retain its capabilities when it switches all of its UIDs to a non-zero value. If this flag is not set, then such a UID switch causes the thread to lose all capabilities. This flag is always cleared on an **execve**(2). (This flag provides the same functionality as the older **prctl**(2) **PR_SET_KEEPCAPS** operation.)

SECURE_NO_SETUID_FIXUP
Setting this flag stops the kernel from adjusting capability sets when the threads's effective and file system UIDs are switched between zero and non-zero values. (See the subsection *Effect of User ID Changes on Capabilities*.)

SECURE_NOROOT
If this bit is set, then the kernel does not grant capabilities when a set-user-ID-root program is executed, or when a process with an effective or real UID of 0 calls **execve**(2). (See the subsection *Capabilities and execution of programs by root*.)

Each of the above "base" flags has a companion "locked" flag. Setting any of the "locked" flags is irreversible, and has the effect of preventing further changes to the corresponding "base" flag. The locked flags are: **SECURE_KEEP_CAPS_LOCKED**, **SECURE_NO_SETUID_FIXUP_LOCKED**, and **SECURE_NOROOT_LOCKED**.

The *securebits* flags can be modified and retrieved using the **prctl**(2) **PR_SET_SECUREBITS** and **PR_GET_SECUREBITS** operations. The **CAP_SETPCAP** capability is required to modify the flags.

The *securebits* flags are inherited by child processes. During an **execve**(2), all of the flags are preserved, except **SECURE_KEEP_CAPS** which is always cleared.

An application can use the following call to lock itself, and all of its descendants, into an environment where the only way of gaining capabilities is by executing a program with associated file capabilities:

```
prctl(PR_SET_SECUREBITS,
      1 << SECURE_KEEP_CAPS_LOCKED |
      1 << SECURE_NO_SETUID_FIXUP |
      1 << SECURE_NO_SETUID_FIXUP_LOCKED |
      1 << SECURE_NOROOT |
      1 << SECURE_NOROOT_LOCKED);
```

CONFORMING TO

No standards govern capabilities, but the Linux capability implementation is based on the withdrawn POSIX.1e draft standard; see *http://wt.xpilot.org/publications/posix.1e/* .

NOTES

Since kernel 2.5.27, capabilities are an optional kernel component, and can be enabled/disabled via the CONFIG_SECURITY_CAPABILITIES kernel configuration option.

The */proc/PID/task/TID/status* file can be used to view the capability sets of a thread. The */proc/PID/status* file shows the capability sets of a process's main thread.

The *libcap* package provides a suite of routines for setting and getting capabilities that is more comfortable and less likely to change than the interface provided by **capset**(2) and **capget**(2). This package also provides the **setcap**(8) and **getcap**(8) programs. It can be found at *http://www.kernel.org/pub/linux/libs/security/linux-privs*.

Before kernel 2.6.24, and since kernel 2.6.24 if file capabilities are not enabled, a thread with the **CAP_SETPCAP** capability can manipulate the capabilities of threads other than itself. However, this is only theoretically possible, since no thread ever has **CAP_SETPCAP** in either of these cases:

* In the pre-2.6.25 implementation the system-wide capability bounding set, */proc/sys/kernel/cap-bound*, always masks out this capability, and this can not be changed without modifying the kernel source and

rebuilding.

* If file capabilities are disabled in the current implementation, then **init** starts out with this capability removed from its per-process bounding set, and that bounding set is inherited by all other processes created on the system.

SEE ALSO

capget(2), **prctl**(2), **setfsuid**(2), **cap_clear**(3), **cap_copy_ext**(3), **cap_from_text**(3), **cap_get_file**(3), **cap_get_proc**(3), **cap_init**(3), **capgetp**(3), **capsetp**(3), **credentials**(7), **pthreads**(7), **getcap**(8), **setcap**(8)

include/linux/capability.h in the kernel source

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.