capabilities(7) — Linux manual page

NAME | DESCRIPTION | CONFORMING TO | NOTES | SEE ALSO | COLOPHON

CAPABILITIES(7) Linux Programmer's Manual CAPABILITIES(7)

NAME top

capabilities - overview of Linux capabilities

DESCRIPTION top

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 nonzero). 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_READ (since Linux 3.16)
Allow reading the audit log via a multicast netlink socket.

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

CAP_BLOCK_SUSPEND (since Linux 3.5)

Employ features that can block system suspend (epoll(7) **EPOLLWAKEUP**, */proc/sys/wake_lock*).

CAP_BPF (since Linux 5.8)
Employ privileged BPF operations; see bpf(2) and bpf-helpers(7).

This capability was added in Linux 5.8 to separate out BPF functionality from the overloaded **CAP_SYS_ADMIN** capability.

CAP_CHECKPOINT_RESTORE (since Linux 5.9)
* Update */proc/sys/kernel/ns_last_pid* (see pid_namespaces(7));
* employ the *set_tid* feature of clone3(2);
* read the contents of the symbolic links in */proc/[pid]/map_files* for other processes.

This capability was added in Linux 5.9 to separate out checkpoint/restore functionality from the overloaded **CAP_SYS_ADMIN** capability.

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;
* invoke open_by_handle_at(2);
* use the linkat(2) **AT_EMPTY_PATH** flag to create a link to a file referred to by a file descriptor.

CAP_FOWNER
* Bypass permission checks on operations that normally require the filesystem 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 inode flags (see ioctl_iflags(2)) on arbitrary files;
* set Access Control Lists (ACLs) on arbitrary files;
* ignore directory sticky bit on file deletion;
* modify *user* extended attributes on sticky directory owned by any user;
* specify **O_NOATIME** for arbitrary files in open(2) and fcntl(2).

CAP_FSETID
* Don't clear set-user-ID and set-group-ID mode bits when a file is modified;
* set the set-group-ID bit for a file whose GID does not match the filesystem 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** inode flags (see ioctl_iflags(2)).

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

CAP_MAC_OVERRIDE (since Linux 2.6.25)
Override Mandatory Access Control (MAC). 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:
* interface configuration;
* administration of IP firewall, masquerading, and accounting;
* modify routing tables;
* bind to any address for transparent proxying;
* set type-of-service (TOS);
* clear driver statistics;
* set promiscuous mode;
* enabling multicasting;
* use setsockopt(2) to set the following socket options: **SO_DEBUG**, **SO_MARK**, **SO_PRIORITY** (for a priority outside the range 0 to 6), **SO_RCVBUFFORCE**, and **SO_SNDBUFFORCE**.

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;
* bind to any address for transparent proxying.

CAP_PERFMON (since Linux 5.8)
Employ various performance-monitoring mechanisms, including:

* call perf_event_open(2);
* employ various BPF operations that have performance implications.

This capability was added in Linux 5.8 to separate out performance monitoring functionality from the overloaded **CAP_SYS_ADMIN** capability. See also the kernel source file *Documentation/admin-guide/perf-security.rst*.

CAP_SETGID
* Make arbitrary manipulations of process GIDs and supplementary GID list;
* forge GID when passing socket credentials via UNIX domain sockets;
* write a group ID mapping in a user namespace (see user_namespaces(7)).

CAP_SETFCAP (since Linux 2.6.24)
Set arbitrary capabilities on a file.

CAP_SETPCAP
If file capabilities are supported (i.e., since Linux 2.6.24): 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.

If file capabilities are not supported (i.e., kernels before Linux 2.6.24): 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.)

CAP_SETUID

* Make arbitrary manipulations of process UIDs (setuid(2), setreuid(2), setresuid(2), setfsuid(2));
* forge UID when passing socket credentials via UNIX domain sockets;
* write a user ID mapping in a user namespace (see user_namespaces(7)).

CAP_SYS_ADMIN

Note: this capability is overloaded; see *Notes to kernel developers*, below.

* Perform a range of system administration operations including: quotactl(2), mount(2), umount(2), pivot_root(2), swapon(2), swapoff(2), sethostname(2), and setdomainname(2);
* perform privileged syslog(2) operations (since Linux 2.6.37, **CAP_SYSLOG** should be used to permit such operations);
* perform **VM86_REQUEST_IRQ** **vm86**(2) command;
* access the same checkpoint/restore functionality that is governed by **CAP_CHECKPOINT_RESTORE** (but the latter, weaker capability is preferred for accessing that functionality).
* perform the same BPF operations as are governed by **CAP_BPF** (but the latter, weaker capability is preferred for accessing that functionality).
* employ the same performance monitoring mechanisms as are governed by **CAP_PERFMON** (but the latter, weaker capability is preferred for accessing that functionality).
* perform **IPC_SET** and **IPC_RMID** operations on arbitrary System V IPC objects;
* override **RLIMIT_NPROC** resource limit;
* perform operations on *trusted* and *security* extended attributes (see xattr(7));
* 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 PID when passing socket credentials via UNIX domain sockets;
* 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_*** flags that create new namespaces with clone(2) and unshare(2) (but, since Linux 3.8, creating user namespaces does not require any capability);
* access privileged *perf* event information;
* call setns(2) (requires **CAP_SYS_ADMIN** in the *target* namespace);
* call fanotify_init(2);
* perform privileged **KEYCTL_CHOWN** and **KEYCTL_SETPERM**

keyctl(2) operations;
* perform madvise(2) **MADV_HWPOISON** operation;
* employ the **TIOCSTI ioctl**(2) to insert characters into the input queue of a terminal other than the caller's controlling terminal;
* employ the obsolete nfsservctl(2) system call;
* employ the obsolete bdflush(2) system call;
* perform various privileged block-device ioctl(2) operations;
* perform various privileged filesystem ioctl(2) operations;
* perform privileged ioctl(2) operations on the */dev/random* device (see random(4));
* install a seccomp(2) filter without first having to set the *no_new_privs* thread attribute;
* modify allow/deny rules for device control groups;
* employ the ptrace(2) **PTRACE_SECCOMP_GET_FILTER** operation to dump tracee's seccomp filters;
* employ the ptrace(2) **PTRACE_SETOPTIONS** operation to suspend the tracee's seccomp protections (i.e., the **PTRACE_O_SUSPEND_SECCOMP** flag);
* perform administrative operations on many device drivers;
* modify autogroup nice values by writing to */proc/[pid]/autogroup* (see sched(7)).

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

CAP_SYS_CHROOT
* Use chroot(2);
* change mount namespaces using setns(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
* Lower the 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), sched_setattr(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);
           * apply get_robust_list(2) to arbitrary processes;
           * transfer data to or from the memory of arbitrary
             processes using process_vm_readv(2) and
             process_vm_writev(2);
           * inspect processes using kcmp(2).

       CAP_SYS_RAWIO
           * Perform I/O port operations (iopl(2) and ioperm(2));
           * access /proc/kcore;
           * employ the FIBMAP ioctl(2) operation;
           * open devices for accessing x86 model-specific registers
             (MSRs, see msr(4));
           * update /proc/sys/vm/mmap_min_addr;
           * create memory mappings at addresses below the value
             specified by /proc/sys/vm/mmap_min_addr;
           * map files in /proc/bus/pci;
           * open /dev/mem and /dev/kmem;
           * perform various SCSI device commands;
           * perform certain operations on hpsa(4) and cciss(4)
             devices;
           * perform a range of device-specific operations on other
             devices.

       CAP_SYS_RESOURCE
           * Use reserved space on ext2 filesystems;
           * make ioctl(2) calls controlling ext3 journaling;
           * override disk quota limits;
           * increase resource limits (see setrlimit(2));
           * override RLIMIT_NPROC resource limit;
           * override maximum number of consoles on console
             allocation;
           * override maximum number of keymaps;
           * allow more than 64hz interrupts from the real-time
             clock;
           * raise msg_qbytes limit for a System V message queue
             above the limit in /proc/sys/kernel/msgmnb (see msgop(2)
             and msgctl(2));
           * allow the RLIMIT_NOFILE resource limit on the number of
             "in-flight" file descriptors to be bypassed when passing
             file descriptors to another process via a UNIX domain
             socket (see unix(7));
           * override the /proc/sys/fs/pipe-size-max limit when
             setting the capacity of a pipe using the F_SETPIPE_SZ
```

fcntl(2) command;
* use **F_SETPIPE_SZ** to increase the capacity of a pipe above the limit specified by */proc/sys/fs/pipe-max-size*;
* override */proc/sys/fs/mqueue/queues_max,* */proc/sys/fs/mqueue/msg_max,* and */proc/sys/fs/mqueue/msgsize_max* limits when creating POSIX message queues (see mq_overview(7));
* employ the prctl(2) **PR_SET_MM** operation;
* set */proc/[pid]/oom_score_adj* to a value lower than the value last set by a process with **CAP_SYS_RESOURCE**.

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

CAP_SYS_TTY_CONFIG
Use vhangup(2); employ various privileged ioctl(2) operations on virtual terminals.

CAP_SYSLOG (since Linux 2.6.37)
* Perform privileged syslog(2) operations. See syslog(2) for information on which operations require privilege.
* View kernel addresses exposed via */proc* and other interfaces when */proc/sys/kernel/kptr_restrict* has the value 1. (See the discussion of the *kptr_restrict* in proc(5).)

CAP_WAKE_ALARM (since Linux 3.0)
Trigger something that will wake up the system (set **CLOCK_REALTIME_ALARM** and **CLOCK_BOOTTIME_ALARM** timers).

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 filesystem 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.

Notes to kernel developers

When adding a new kernel feature that should be governed by a capability, consider the following points.

* The goal of capabilities is divide the power of superuser into

pieces, such that if a program that has one or more capabilities is compromised, its power to do damage to the system would be less than the same program running with root privilege.

* You have the choice of either creating a new capability for your new feature, or associating the feature with one of the existing capabilities. In order to keep the set of capabilities to a manageable size, the latter option is preferable, unless there are compelling reasons to take the former option. (There is also a technical limit: the size of capability sets is currently limited to 64 bits.)

* To determine which existing capability might best be associated with your new feature, review the list of capabilities above in order to find a "silo" into which your new feature best fits. One approach to take is to determine if there are other features requiring capabilities that will always be used along with the new feature. If the new feature is useless without these other features, you should use the same capability as the other features.

* *Don't* choose **CAP_SYS_ADMIN** if you can possibly avoid it! A vast proportion of existing capability checks are associated with this capability (see the partial list above). It can plausibly be called "the new root", since on the one hand, it confers a wide range of powers, and on the other hand, its broad scope means that this is the capability that is required by many privileged programs. Don't make the problem worse. The only new features that should be associated with **CAP_SYS_ADMIN** are ones that *closely* match existing uses in that silo.

* If you have determined that it really is necessary to create a new capability for your feature, don't make or name it as a "single-use" capability. Thus, for example, the addition of the highly specific **CAP_SYS_PACCT** was probably a mistake. Instead, try to identify and name your new capability as a broader silo into which other related future use cases might fit.

Thread capability sets

Each thread has the following 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 reacquire 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). Inheritable capabilities remain inheritable when executing any program, and inheritable capabilities are added to the permitted set when executing a program that has the corresponding bits set in the file inheritable set.

Because inheritable capabilities are not generally preserved across execve(2) when running as a non-root user, applications that wish to run helper programs with elevated capabilities should consider using ambient capabilities, described below.

Effective

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

Bounding (per-thread since Linux 2.6.25)

The capability bounding set is a mechanism that can be used to limit the capabilities that are gained during execve(2).

Since Linux 2.6.25, this is a per-thread capability set. In older kernels, the capability bounding set was a system wide attribute shared by all threads on the system.

For more details on the capability bounding set, see below.

Ambient (since Linux 4.3)

This is a set of capabilities that are preserved across an execve(2) of a program that is not privileged. The ambient capability set obeys the invariant that no capability can ever be ambient if it is not both permitted and inheritable.

The ambient capability set can be directly modified using prctl(2). Ambient capabilities are automatically lowered if either of the corresponding permitted or inheritable capabilities is lowered.

Executing a program that changes UID or GID due to the set-user-ID or set-group-ID bits or executing a program that has any file capabilities set will clear the ambient set. Ambient capabilities are added to the permitted set and assigned to the effective set when execve(2) is called. If ambient capabilities cause a process's permitted and effective capabilities to increase during an execve(2), this does not trigger the secure-execution mode described in ld.so(8).

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).

Since Linux 3.2, the file */proc/sys/kernel/cap_last_cap* exposes the numerical value of the highest capability supported by the running kernel; this can be used to determine the highest bit that may be set in a capability set.

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) and xattr(7)) 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.

File capability extended attribute versioning

To allow extensibility, the kernel supports a scheme to encode a version number inside the *security.capability* extended attribute that is used to implement file capabilities. These version numbers are internal to the implementation, and not directly visible to user-space applications. To date, the following versions are supported:

VFS_CAP_REVISION_1
This was the original file capability implementation, which supported 32-bit masks for file capabilities.

VFS_CAP_REVISION_2 (since Linux 2.6.25)
This version allows for file capability masks that are 64 bits in size, and was necessary as the number of supported capabilities grew beyond 32. The kernel transparently continues to support the execution of files that have 32-bit version 1 capability masks, but when adding capabilities to files that did not previously have capabilities, or modifying the capabilities of existing files, it automatically uses the version 2 scheme (or possibly the version 3 scheme, as described below).

VFS_CAP_REVISION_3 (since Linux 4.14)
Version 3 file capabilities are provided to support namespaced file capabilities (described below).

As with version 2 file capabilities, version 3 capability masks are 64 bits in size. But in addition, the root user ID of namespace is encoded in the *security.capability* extended attribute. (A namespace's root user ID is the value that user ID 0 inside that namespace maps to in the initial user namespace.)

Version 3 file capabilities are designed to coexist with version 2 capabilities; that is, on a modern Linux system, there may be some files with version 2 capabilities while others have version 3 capabilities.

Before Linux 4.14, the only kind of file capability extended attribute that could be attached to a file was a **VFS_CAP_REVISION_2** attribute. Since Linux 4.14, the version of the *security.capability* extended attribute that is attached to a file depends on the circumstances in which the attribute was created.

Starting with Linux 4.14, a *security.capability* extended attribute is automatically created as (or converted to) a version 3 (**VFS_CAP_REVISION_3**) attribute if both of the following are true:

(1) The thread writing the attribute resides in a noninitial user namespace. (More precisely: the thread resides in a user namespace other than the one from which the underlying filesystem was mounted.)

(2) The thread has the **CAP_SETFCAP** capability over the file inode, meaning that (a) the thread has the **CAP_SETFCAP** capability in its own user namespace; and (b) the UID and GID of the file inode have mappings in the writer's user namespace.

When a **VFS_CAP_REVISION_3** *security.capability* extended attribute is created, the root user ID of the creating thread's user namespace is saved in the extended attribute.

By contrast, creating or modifying a *security.capability* extended attribute from a privileged (**CAP_SETFCAP**) thread that resides in the namespace where the underlying filesystem was mounted (this normally means the initial user namespace) automatically results in the creation of a version 2 (**VFS_CAP_REVISION_2**) attribute.

Note that the creation of a version 3 *security.capability* extended attribute is automatic. That is to say, when a user-space application writes (setxattr(2)) a *security.capability* attribute in the version 2 format, the kernel will automatically create a version 3 attribute if the attribute is created in the circumstances described above. Correspondingly, when a version 3 *security.capability* attribute is retrieved (getxattr(2)) by a process that resides inside a user namespace that was created by the root user ID (or a descendant of that user namespace), the returned attribute is (automatically) simplified to appear as a version 2 attribute (i.e., the returned value is the size of a version 2 attribute and does not include the root user ID). These automatic translations mean that no changes are required to user-space tools (e.g., **setcap**(1) and **getcap**(1)) in order for those tools to be used to create and retrieve version 3 *security.capability* attributes.

Note that a file can have either a version 2 or a version 3 *security.capability* extended attribute associated with it, but not both: creation or modification of the *security.capability* extended attribute will automatically modify the version according to the circumstances in which the extended attribute is created or modified.

Transformation of capabilities during execve()

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

```
P'(ambient)     = (file is privileged) ? 0 : P(ambient)

P'(permitted)   = (P(inheritable) & F(inheritable)) |
                  (F(permitted) & P(bounding)) | P'(ambient)

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

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

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

where:

P() denotes the value of a thread capability set before the execve(2)

P'() denotes the value of a thread capability set after the execve(2)

F() denotes a file capability set

Note the following details relating to the above capability transformation rules:

* The ambient capability set is present only since Linux 4.3. When determining the transformation of the ambient set during execve(2), a privileged file is one that has capabilities or has the set-user-ID or set-group-ID bit set.

* Prior to Linux 2.6.25, the bounding set was a system-wide attribute shared by all threads. That system-wide value was employed to calculate the new permitted set during execve(2) in the same manner as shown above for *P(bounding)*.

Note: during the capability transitions described above, file capabilities may be ignored (treated as empty) for the same reasons that the set-user-ID and set-group-ID bits are ignored; see execve(2). File capabilities are similarly ignored if the kernel was booted with the *no_file_caps* option.

Note: according to the rules above, if a process with nonzero user IDs performs an execve(2) then any capabilities that are present in its permitted and effective sets will be cleared. For the treatment of capabilities when a process with a user ID of zero performs an execve(2), see below under *Capabilities and execution of programs by root*.

Safety checking for capability-dumb binaries

A capability-dumb binary is an application that has been marked to have file capabilities, but has not been converted to use the libcap(3) API to manipulate its capabilities. (In other words, this is a traditional set-user-ID-root program that has been switched to use file capabilities, but whose code has not been modified to understand capabilities.) For such applications, the effective capability bit is set on the file, so that the file permitted capabilities are automatically enabled in the process effective set when executing the file. The kernel recognizes a file which has the effective capability bit set as capability-dumb for the purpose of the check described here.

When executing a capability-dumb binary, the kernel checks if the process obtained all permitted capabilities that were specified in the file permitted set, after the capability transformations described above have been performed. (The typical reason why this might *not* occur is that the capability bounding set masked out some of the capabilities in the file permitted set.) If the process did not obtain the full set of file permitted capabilities, then execve(2) fails with the error **EPERM**. This prevents possible security risks that could arise when a capability-dumb application is executed with less privilege that it needs. Note that, by definition, the application could not itself recognize this problem, since it does not employ the libcap(3) API.

Capabilities and execution of programs by root

In order to mirror traditional UNIX semantics, the kernel performs special treatment of file capabilities when a process with UID 0 (root) executes a program and when a set-user-ID-root program is executed.

After having performed any changes to the process effective ID that were triggered by the set-user-ID mode bit of the binary—e.g., switching the effective user ID to 0 (root) because a set-user-ID-root program was executed—the kernel calculates the file capability sets as follows:

1. If the real or effective user ID of the process is 0 (root), then the file inheritable and permitted sets are ignored; instead they are notionally considered to be all ones (i.e., all capabilities enabled). (There is one exception to this behavior, described below in *Set-user-ID-root programs that have file capabilities*.)

2. If the effective user ID of the process is 0 (root) or the file effective bit is in fact enabled, then the file effective bit is notionally defined to be one (enabled).

These notional values for the file's capability sets are then used as described above to calculate the transformation of the process's capabilities during execve(2).

Thus, when a process with nonzero UIDs execve(2)s a set-user-ID-root program that does not have capabilities attached, or when a process whose real and effective UIDs are zero execve(2)s a program, the calculation of the process's new permitted capabilities simplifies to:

```
P'(permitted)   = P(inheritable) | P(bounding)

P'(effective)   = P'(permitted)
```

Consequently, the process gains all capabilities in its permitted and effective capability sets, except those masked out by the capability bounding set. (In the calculation of P'(permitted), the P'(ambient) term can be simplified away because it is by definition a proper subset of P(inheritable).)

The special treatments of user ID 0 (root) described in this subsection can be disabled using the securebits mechanism described below.

Set-user-ID-root programs that have file capabilities

There is one exception to the behavior described under *Capabilities and execution of programs by root*. If (a) the binary that is being executed has capabilities attached and (b) the real user ID of the process is *not* 0 (root) and (c) the effective user ID of the process *is* 0 (root), then the file capability bits are honored (i.e., they are not notionally considered to be all ones). The usual way in which this situation can arise is when executing a set-UID-root program that also has file capabilities. When such a program is executed, the process gains just the capabilities granted by the program (i.e., not all capabilities, as would occur when executing a set-user-ID-root program that does not have any associated file capabilities).

Note that one can assign empty capability sets to a program file, and thus it is possible to create a set-user-ID-root program that changes the effective and saved set-user-ID of the process that executes the program to 0, but confers no capabilities to that process.

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 a capability is not in the bounding set, then a thread can't add this capability to its inheritable set, even if it was in its permitted capabilities, and thereby cannot have this 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 inheritable capabilities. If a thread maintains a capability in its inheritable 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 inheritable set.

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

Capability bounding set from Linux 2.6.25 onward

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

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 supported only if file capabilities are compiled into the kernel. In kernels before Linux 2.6.33, file capabilities were an optional feature configurable via the **CONFIG_SECURITY_FILE_CAPABILITIES** option. Since Linux 2.6.33, the configuration option has been removed and file capabilities are always part of the kernel. When file capabilities are compiled into the kernel, 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 inheritable set. However it does prevent the capability from being added back into the thread's inheritable set in the future.

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: a process 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.

Effect of user ID changes on capabilities

To preserve the traditional semantics for transitions between 0 and nonzero 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 filesystem 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 nonzero value, then all capabilities are cleared from the permitted, effective, and ambient capability sets.

2. If the effective user ID is changed from 0 to nonzero, then all capabilities are cleared from the effective set.

3. If the effective user ID is changed from nonzero to 0, then the permitted set is copied to the effective set.

4. If the filesystem user ID is changed from 0 to nonzero (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**, **CAP_LINUX_IMMUTABLE** (since Linux 2.6.30), **CAP_MAC_OVERRIDE**, and **CAP_MKNOD** (since Linux 2.6.30). If the filesystem UID is changed from nonzero 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 nonzero values, it can do so using the **SECBIT_KEEP_CAPS** securebits flag described below.

Programmatically adjusting capability sets

A thread can retrieve and change its permitted, effective, and inheritable 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 Linux 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:

SECBIT_KEEP_CAPS

Setting this flag allows a thread that has one or more 0 UIDs to retain capabilities in its permitted set when it switches all of its UIDs to nonzero values. If this flag is not set, then such a UID switch causes the thread to lose all permitted capabilities. This flag is always cleared on an execve(2).

Note that even with the **SECBIT_KEEP_CAPS** flag set, the effective capabilities of a thread are cleared when it switches its effective UID to a nonzero value. However, if the thread has set this flag and its effective UID is already nonzero, and the thread subsequently switches all other UIDs to nonzero values, then the effective capabilities will not be cleared.

The setting of the **SECBIT_KEEP_CAPS** flag is ignored if the **SECBIT_NO_SETUID_FIXUP** flag is set. (The latter flag provides a superset of the effect of the former flag.)

This flag provides the same functionality as the older prctl(2) **PR_SET_KEEPCAPS** operation.

SECBIT_NO_SETUID_FIXUP
Setting this flag stops the kernel from adjusting the process's permitted, effective, and ambient capability sets when the thread's effective and filesystem UIDs are switched between zero and nonzero values. (See the subsection *Effect of user ID changes on capabilities*.)

SECBIT_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*.)

SECBIT_NO_CAP_AMBIENT_RAISE
Setting this flag disallows raising ambient capabilities via the prctl(2) **PR_CAP_AMBIENT_RAISE** operation.

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: **SECBIT_KEEP_CAPS_LOCKED**, **SECBIT_NO_SETUID_FIXUP_LOCKED**, **SECBIT_NOROOT_LOCKED**, and **SECBIT_NO_CAP_AMBIENT_RAISE_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. Note that the **SECBIT_*** constants are available only after including the *<linux/securebits.h>* header file.

The *securebits* flags are inherited by child processes. During an execve(2), all of the flags are preserved, except **SECBIT_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,
        /* SECBIT_KEEP_CAPS off */
        SECBIT_KEEP_CAPS_LOCKED |
        SECBIT_NO_SETUID_FIXUP |
        SECBIT_NO_SETUID_FIXUP_LOCKED |
        SECBIT_NOROOT |
        SECBIT_NOROOT_LOCKED);
```

```
/* Setting/locking SECBIT_NO_CAP_AMBIENT_RAISE
   is not required */
```

Per-user-namespace "set-user-ID-root" programs

A set-user-ID program whose UID matches the UID that created a user namespace will confer capabilities in the process's permitted and effective sets when executed by any process inside that namespace or any descendant user namespace.

The rules about the transformation of the process's capabilities during the execve(2) are exactly as described in the subsections *Transformation of capabilities during execve()* and *Capabilities and execution of programs by root*, with the difference that, in the latter subsection, "root" is the UID of the creator of the user namespace.

Namespaced file capabilities

Traditional (i.e., version 2) file capabilities associate only a set of capability masks with a binary executable file. When a process executes a binary with such capabilities, it gains the associated capabilities (within its user namespace) as per the rules described above in "Transformation of capabilities during execve()".

Because version 2 file capabilities confer capabilities to the executing process regardless of which user namespace it resides in, only privileged processes are permitted to associate capabilities with a file. Here, "privileged" means a process that has the **CAP_SETFCAP** capability in the user namespace where the filesystem was mounted (normally the initial user namespace). This limitation renders file capabilities useless for certain use cases. For example, in user-namespaced containers, it can be desirable to be able to create a binary that confers capabilities only to processes executed inside that container, but not to processes that are executed outside the container.

Linux 4.14 added so-called namespaced file capabilities to support such use cases. Namespaced file capabilities are recorded as version 3 (i.e., **VFS_CAP_REVISION_3**) *security.capability* extended attributes. Such an attribute is automatically created in the circumstances described above under "File capability extended attribute versioning". When a version 3 *security.capability* extended attribute is created, the kernel records not just the capability masks in the extended attribute, but also the namespace root user ID.

As with a binary that has **VFS_CAP_REVISION_2** file capabilities, a binary with **VFS_CAP_REVISION_3** file capabilities confers capabilities to a process during **execve**(). However, capabilities are conferred only if the binary is executed by a process that resides in a user namespace whose UID 0 maps to the root user ID that is saved in the extended attribute, or when executed by a process that resides in a descendant of such a namespace.

Interaction with user namespaces

For further information on the interaction of capabilities and user namespaces, see user_namespaces(7).

CONFORMING TO top

No standards govern capabilities, but the Linux capability implementation is based on the withdrawn POSIX.1e draft standard; see ⟨https://archive.org/details/posix_1003.1e-990310⟩.

NOTES top

When attempting to strace(1) binaries that have capabilities (or set-user-ID-root binaries), you may find the *-u <username>* option useful. Something like:

```
$ sudo strace -o trace.log -u ceci ./myprivprog
```

From kernel 2.5.27 to kernel 2.6.26, capabilities were an optional kernel component, and could 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. Before Linux 3.8, nonexistent capabilities were shown as being enabled (1) in these sets. Since Linux 3.8, all nonexistent capabilities (above **CAP_LAST_CAP**) are shown as disabled (0).

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 ⟨https://git.kernel.org/pub/scm/libs/libcap/libcap.git/refs/⟩.

Before kernel 2.6.24, and from kernel 2.6.24 to kernel 2.6.32 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 the **CAP_SETPCAP** capability, and this can not be changed without modifying the kernel source and rebuilding the kernel.

* If file capabilities are disabled (i.e., the kernel

CONFIG_SECURITY_FILE_CAPABILITIES option is disabled), then **init** starts out with the **CAP_SETPCAP** capability removed from its per-process bounding set, and that bounding set is inherited by all other processes created on the system.

SEE ALSO top

capsh(1), setpriv(1), 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), libcap(3), proc(5), credentials(7), pthreads(7), user_namespaces(7), captest(8), filecap(8), getcap(8), getpcaps(8), netcap(8), pscap(8), setcap(8)

include/linux/capability.h in the Linux kernel source tree

COLOPHON top

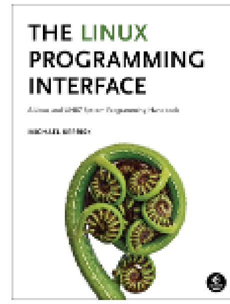
This page is part of release 5.10 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Linux **2020-08-13** **CAPABILITIES(7)**

Pages that refer to this page: capsh(1), setpriv(1), systemd-analyze(1), systemd-nspawn(1), adjtimex(2), capget(2), clone(2), execve(2), fcntl(2), fork(2), getgroups(2), getpriority(2), getrlimit(2), gettimeofday(2), intro(2), ioperm(2), iopl(2), ioprio_set(2), keyctl(2), kill(2), mlock(2), msgctl(2), msgget(2), msgop(2), nice(2), pciconfig_read(2), prctl(2), ptrace(2), reboot(2), request_key(2), sched_setaffinity(2), sched_setattr(2), sched_setparam(2), sched_setscheduler(2), semctl(2), semget(2), semop(2), seteuid(2), setfsgid(2), setfsuid(2), setgid(2), setresuid(2), setreuid(2), setuid(2), shmctl(2), shmget(2), shmop(2), spu_create(2), spu_run(2), stat(2), statx(2), stime(2), syslog(2), uselib(2), vhangup(2), cap_clear(3), cap_copy_ext(3), cap_from_text(3), cap_get_file(3), cap_get_proc(3), cap_init(3), capng_apply(3), capng_apply_caps_fd(3), capng_capability_to_name(3), capng_change_id(3), capng_clear(3), capng_fill(3), capng_get_caps_fd(3), capng_get_caps_process(3), capng_get_rootid(3), capng_have_capabilities(3), capng_have_capability(3), capng_lock(3), capng_name_to_capability(3), capng_restore_state(3), capng_save_state(3), capng_setpid(3), capng_set_rootid(3), capng_update(3), capng_updatev(3), getauxval(3), getenv(3), intro(3), killpg(3), libcap(3), pthread_create(3), sd_bus_creds_get_pid(3), sd_bus_query_sender_creds(3), core(5), lxc.container.conf(5), proc(5), systemd.exec(5), systemd.nspawn(5), systemd-system.conf(5), systemd.unit(5), arp(7), credentials(7), ddp(7), ip(7), libdrop_ambient(7), namespaces(7), netdevice(7), netlink(7), packet(7), path_resolution(7), pid_namespaces(7), pthreads(7), raw(7), sched(7), socket(7), spufs(7), systemd.journal-fields(7), unix(7), user_namespaces(7), vsock(7), xattr(7), captest(8), filecap(8), getcap(8), getpcaps(8), ld.so(8), mount.fuse3(8), netcap(8), pscap(8), setcap(8)

HTML rendering created 2020-12-21 by Michael Kerrisk, author of *The Linux*

Programming Interface, maintainer of the Linux *man-pages* project.



For details of in-depth **Linux/UNIX system programming training courses** that I teach, look here.

Hosting by jambit GmbH.