39
CAPABILITIES

This chapter describes the Linux capabilities scheme, which divides the traditional all-or-nothing UNIX privilege scheme into individual capabilities that can be independently enabled or disabled. Using capabilities allows a program to perform some privileged operations, while preventing it from performing others.

39.1 Rationale for Capabilities

The traditional UNIX privilege scheme divides processes into two categories: those whose effective user ID is 0 (superuser), which bypass all privilege checks, and all other processes, which are subject to privilege checking according to their user and group IDs.

The coarse granularity of this scheme is a problem. If we want to allow a process to perform some operation that is permitted only to the superuser—for example, changing the system time—then we must run that process with an effective user ID of 0. (If an unprivileged user needs to perform such operations, this is typically implemented using a set-user-ID-*root* program.) However, this grants the process privileges to perform a host of other actions as well—for example, bypassing all permission checks when accessing files—thus opening the door for a range of security breaches if the program behaves in unexpected ways (which may be the consequence of unforeseen circumstances, or because of deliberate manipulation by a malicious user). The traditional way of dealing with this problem was outlined in Chapter 38: we drop effective privileges (i.e., change from an effective user ID of 0, while maintaining 0 in the saved set-user-ID) and temporarily reacquire them only when needed.

The Linux capability scheme refines the handling of this problem. Rather than using a single privilege (i.e., effective user ID of 0) when performing security checks in the kernel, the superuser privilege is divided into distinct units, called *capabilities*. Each privileged operation is associated with a particular capability, and a process can perform that operation only if it has the corresponding capability (regardless of its effective user ID). Put another way, everywhere in this book that we talk about a privileged process on Linux, what we really mean is a process that has the relevant capability for performing a particular operation.

Most of the time, the Linux capability scheme is invisible to us. The reason for this is that when an application that is unaware of capabilities assumes an effective user ID of 0, the kernel grants that process the complete range of capabilities.

The Linux capabilities implementation is based on the POSIX 1003.1e draft standard (*http://wt.tuxomania.net/publications/posix.1e/*). This standardization effort foundered in the late 1990s before it was completed, but various capabilities implementations are nevertheless based on the draft standard. (Some of the capabilities listed in Table 39-1 are defined in the POSIX.1e draft, but many are Linux extensions.)

> Capability schemes are provided in a few other UNIX implementations, such as in Sun's Solaris 10 and earlier Trusted Solaris releases, SGI's Trusted Irix, and as part of the TrustedBSD project for FreeBSD ([Watson, 2000]). Similar schemes exist in some other operating systems; for example, the privilege mechanism in Digital's VMS system.

39.2 The Linux Capabilities

Table 39-1 lists the Linux capabilities and provides an abbreviated (and incomplete) guide to the operations to which they apply.

39.3 Process and File Capabilities

Each process has three associated capability sets—termed *permitted*, *effective*, and *inheritable*—that can contain zero or more of the capabilities listed in Table 39-1. Each file can likewise have three associated capability sets, with the same names. (For reasons that will become evident, the file effective capability set is really just a single bit that is either enabled or disabled.) We go into the details of each of these capability sets in the following sections.

39.3.1 Process Capabilities

For each process, the kernel maintains three capability sets (implemented as bit masks) in which zero or more of the capabilities specified in Table 39-1 are enabled. The three sets are as follows:

- *Permitted*: These are the capabilities that a process *may* employ. The permitted set is a limiting superset for the capabilities that can be added to the effective and inheritable sets. If a process drops a capability from its permitted set, it can never reacquire that capability (unless it execs a program that once more confers the capability).

- *Effective*: These are the capabilities used by the kernel to perform privilege checking for the process. As long as it maintains a capability in its permitted set, a process can temporarily disable the capability by dropping it from the effective set, and then later restoring it to that set.
- *Inheritable*: These are capabilities that may be carried over to the permitted set when a program is execed by this process.

We can view hexadecimal representations of the three capability sets for any process in the three fields `CapInh`, `CapPrm`, and `CapEff` in the Linux-specific `/proc/`*PID*`/status` file.

> The *getpcap* program (part of the *libcap* package described in Section 39.7) can be used to display the capabilities of a process in an easier-to-read format.

A child process produced via *fork()* inherits copies of its parent's capability sets. We describe the treatment of capability sets during an *exec()* in Section 39.5.

> In reality, capabilities are a per-thread attribute that can be adjusted independently for each of the threads in a process. The capabilities of a specific thread within a multithreaded process are shown in the `/proc/`*PID*`/task/`*TID*`/status` file. The `/proc/`*PID*`/status` file shows the capabilities of the main thread.
>
> Before kernel 2.6.25, Linux represented capability sets using 32 bits. The addition of further capabilities in kernel 2.6.25 required a move to 64-bit sets.

39.3.2 File Capabilities

If a file has associated capability sets, then these sets are used to determine the capabilities that are given to a process if it execs that file. There are three file capability sets:

- *Permitted*: This is a set of capabilities that may be added to the process's permitted set during an *exec()*, regardless of the process's existing capabilities.
- *Effective*: This is just a single bit. If it is enabled, then, during an *exec()*, the capabilities that are enabled in the process's new permitted set are also enabled in the process's new effective set. If the file effective bit is disabled, then, after an *exec()*, the process's new effective set is initially empty.
- *Inheritable*: This set is masked against the process's inheritable set to determine a set of capabilities that are to be enabled in the process's permitted set after an *exec()*.

Section 39.5 provides details of how file capabilities are used during an *exec()*.

> The permitted and inheritable file capabilities were formerly known as *forced* and *allowed*. Those terms are now obsolete, but they are still informative. The permitted file capabilities are the ones that are *forced* into the process's permitted set during an *exec()*, regardless of the process's existing capabilities. The inheritable file capabilities are the ones that the file *allows* into the process's permitted set during an *exec()*, if those capabilities are also enabled in the process's inheritable capability set.
>
> The capabilities associated with a file are stored in a *security* extended attribute (Section 16.1) named *security.capability*. The `CAP_SETFCAP` capability is required to update this extended attribute.

Table 39-1: Operations permitted by each Linux capability

Capability	**Permits process to**
`CAP_AUDIT_CONTROL`	(Since Linux 2.6.11) Enable and disable kernel audit logging; change filtering rules for auditing; retrieve auditing status and filtering rules
`CAP_AUDIT_WRITE`	(Since Linux 2.6.11) Write records to the kernel auditing log
`CAP_CHOWN`	Change file's user ID (owner) or change file's group ID to a group of which process is not a member (*chown()*)
`CAP_DAC_OVERRIDE`	Bypass file read, write, and execute permission checks (DAC is an abbreviation for discretionary access control); read contents of `cwd`, `exe`, and `root` symbolic links in `/proc/`*PID*
`CAP_DAC_READ_SEARCH`	Bypass file read permission checks and directory read and execute (search) permission checks
`CAP_FOWNER`	Generally ignore permission checks on operations that normally require the process's file-system user ID to match the file's user ID (*chmod()*, *utime()*); set i-node flags on arbitrary files; set and modify ACLs on arbitrary files; ignore effect of directory sticky bit when deleting files (*unlink()*, *rmdir()*, *rename()*); specify `O_NOATIME` flag for arbitrary files in *open()* and *fcntl(F_SETFL)*
`CAP_FSETID`	Modify a file without having the kernel turn off set-user-ID and set-group-ID bits (*write()*, *truncate()*); enable set-group-ID bit for a file whose group ID doesn't match the process's file-system group ID or supplementary group IDs (*chmod()*)
`CAP_IPC_LOCK`	Override memory-locking restrictions (*mlock()*, *mlockall()*, *shmctl(SHM_LOCK)*, *shmctl(SHM_UNLOCK)*); employ *shmget()* `SHM_HUGETLB` flag and *mmap()* `MAP_HUGETLB` flag.
`CAP_IPC_OWNER`	Bypass permission checks for operations on System V IPC objects
`CAP_KILL`	Bypass permission checks for sending signals (*kill()*, *sigqueue()*)
`CAP_LEASE`	(Since Linux 2.4) Establish leases on arbitrary files (*fcntl(F_SETLEASE)*)
`CAP_LINUX_IMMUTABLE`	Set append and immutable i-node flags
`CAP_MAC_ADMIN`	(Since Linux 2.6.25) Configure or make state changes for mandatory access control (MAC) (implemented by some Linux security modules)
`CAP_MAC_OVERRIDE`	(Since Linux 2.6.25) Override MAC (implemented by some Linux security modules)
`CAP_MKNOD`	(Since Linux 2.4) Use *mknod()* to create devices
`CAP_NET_ADMIN`	Perform various network-related operations (e.g., setting privileged socket options, enabling multicasting, configuring network interfaces, and modifying routing tables)
`CAP_NET_BIND_SERVICE`	Bind to privileged socket ports
`CAP_NET_BROADCAST`	(Unused) Perform socket broadcasts and listen to multicasts
`CAP_NET_RAW`	Use raw and packet sockets
`CAP_SETGID`	Make arbitrary changes to process group IDs (*setgid()*, *setegid()*, *setregid()*, *setresgid()*, *setfsgid()*, *setgroups()*, *initgroups()*); forge group ID when passing credentials via UNIX domain socket (`SCM_CREDENTIALS`)
`CAP_SETFCAP`	(Since Linux 2.6.24) Set file capabilities

Table 39-1: Operations permitted by each Linux capability (continued)

Capability	**Permits process to**
`CAP_SETPCAP`	If file capabilities are not supported, grant and remove capabilities in the process's permitted set to or from any other process (including self); if file capabilities are supported, add any capability in the process's capability bounding set to its inheritable set, drop capabilities from the bounding set, and change *securebits* flags
`CAP_SETUID`	Make arbitrary changes to process user IDs (*setuid()*, *seteuid()*, *setreuid()*, *setresuid()*, *setfsuid()*); forge user ID when passing credentials via UNIX domain socket (`SCM_CREDENTIALS`)
`CAP_SYS_ADMIN`	Exceed `/proc/sys/fs/file-max` limit in system calls that open files (e.g., *open()*, *shm_open()*, *pipe()*, *socket()*, *accept()*, *exec()*, *acct()*, *epoll_create()*); perform various system administration operations, including *quotactl()* (control disk quotas), *mount()* and *umount()*, *swapon()* and *swapoff()*, *pivot_root()*, *sethostname()* and *setdomainname()*; perform various *syslog(2)* operations; override `RLIMIT_NPROC` resource limit (*fork()*); call *lookup_dcookie()*; set *trusted* and *security* extended attributes; perform `IPC_SET` and `IPC_RMID` operations on arbitrary System V IPC objects; forge process ID when passing credentials via UNIX domain socket (`SCM_CREDENTIALS`); use *ioprio_set()* to assign `IOPRIO_CLASS_RT` scheduling class; employ `TIOCCONS` *ioctl()*; employ `CLONE_NEWNS` flag with *clone()* and *unshare()*; perform `KEYCTL_CHOWN` and `KEYCTL_SETPERM` *keyctl()* operations; administer *random(4)* device; various device-specific operations
`CAP_SYS_BOOT`	Use *reboot()* to reboot the system; call *kexec_load()*
`CAP_SYS_CHROOT`	Use *chroot()* to set process root directory
`CAP_SYS_MODULE`	Load and unload kernel modules (*init_module()*, *delete_module()*, *create_module()*)
`CAP_SYS_NICE`	Raise nice value (*nice()*, *setpriority()*); change nice value for arbitrary processes (*setpriority()*); set `SCHED_RR` and `SCHED_FIFO` realtime scheduling policies for calling process; reset `SCHED_RESET_ON_FORK` flag; set scheduling policies and priorities for arbitrary processes (*sched_setscheduler()*, *sched_setparam()*); set I/O scheduling class and priority for arbitrary processes (*ioprio_set()*); set CPU affinity for arbitrary processes (*sched_setaffinity()*); use *migrate_pages()* to migrate arbitrary processes and allow processes to be migrated to arbitrary nodes; apply *move_pages()* to arbitrary processes; use `MPOL_MF_MOVE_ALL` flag with *mbind()* and *move_pages()*
`CAP_SYS_PACCT`	Use *acct()* to enable or disable process accounting
`CAP_SYS_PTRACE`	Trace arbitrary processes using *ptrace()*; access `/proc/`*PID*`/environ` for arbitrary processes; apply *get_robust_list()* to arbitrary processes
`CAP_SYS_RAWIO`	Perform operations on I/O ports using *iopl()* and *ioperm()*; access `/proc/kcore`; open `/dev/mem` and `/dev/kmem`
`CAP_SYS_RESOURCE`	Use reserved space on file systems; make *ioctl()* calls controlling *ext3* journaling; override disk quota limits; increase hard resource limits (*setrlimit()*); override `RLIMIT_NPROC` resource limit (*fork()*); raise *msg_qbytes* limit for a System V message queue above limit in `/proc/sys/kernel/msgmnb`; bypass various POSIX message queue limits defined by files under `/proc/sys/fs/mqueue`
`CAP_SYS_TIME`	Modify system clock (*settimeofday()*, *stime()*, *adjtime()*, *adjtimex()*); set hardware clock
`CAP_SYS_TTY_CONFIG`	Perform virtual hangup of terminal or pseudoterminal using *vhangup()*

39.3.3 Purpose of the Process Permitted and Effective Capability Sets

The *process permitted* capability set defines the capabilities that a process *may* employ. The *process effective* capability set defines the capabilities that are currently in effect for the process—that is, the set of capabilities that the kernel uses when checking whether the process has the necessary privilege to perform a particular operation.

The permitted capability set imposes an upper bound on the effective set. A process can *raise* a capability in its effective set only if that capability is in the permitted set. (The terms *add* to and *set* are sometimes used synonymously with *raise*. The converse operation is *drop*, or synonymously, *remove* or *clear*.)

The relationship between the effective and permitted capability sets is analogous to that between the effective user ID and the saved set-user-ID for a set-user-ID-*root* program. Dropping a capability from the effective set is analogous to temporarily dropping an effective user ID of 0, while maintaining 0 in the saved set-user-ID. Dropping a capability from both the effective and permitted capability sets is analogous to permanently dropping superuser privileges by setting both the effective user ID and the saved set-user ID to nonzero values.

39.3.4 Purpose of the File Permitted and Effective Capability Sets

The *file permitted* capability set provides a mechanism by which an executable file can give capabilities to a process. It specifies a group of capabilities that are to be assigned to the process's permitted capability set during an *exec()*.

The *file effective* capability set is a single flag (bit) that is either enabled or disabled. To understand why this set consists of just a single bit, we need to consider the two cases that occur when a program is execed:

- The program may be *capability-dumb*, meaning that it doesn't know about capabilities (i.e., it is designed as a traditional set-user-ID-*root* program). Such a program won't know that it needs to raise capabilities in its effective set in order to be able to perform privileged operations. For such programs, an *exec()* should have the effect that all of the process's new permitted capabilities are automatically also assigned to its effective set. This result is achieved by enabling the file effective bit.
- The program may be *capability-aware*, meaning that it has been designed with the capabilities framework in mind, and it will make the appropriate system calls (discussed later) to raise and drop capabilities in its effective set. For such programs, least-privilege considerations mean that, after an *exec()*, all capabilities should initially be disabled in the process's effective capability set. This result is achieved by disabling the file effective capability bit.

39.3.5 Purpose of the Process and File Inheritable Sets

At first glance, the use of permitted and effective sets for processes and files might seem a sufficient framework for a capabilities system. However, there are some situations where they do not suffice. For example, what if a process performing an *exec()* wants to preserve some of its current capabilities across the *exec()*? It might appear that the capabilities implementation could provide this feature simply by preserving the process's permitted capabilities across an *exec()*. However, this approach would not handle the following cases:

- Performing the *exec()* might require certain privileges (e.g., `CAP_DAC_OVERRIDE`) that we don't want to preserve across the *exec()*.
- Suppose that we explicitly dropped some permitted capabilities that we didn't want to preserve across the *exec()*, but then the *exec()* failed. In this case, the program might need some of the permitted capabilities that it has already (irrevocably) dropped.

For these reasons, a process's permitted capabilities are not preserved across an *exec()*. Instead, another capability set is introduced: the *inheritable set*. The inheritable set provides a mechanism by which a process can preserve some of its capabilities across an *exec()*.

The *process inheritable* capability set specifies a group of capabilities that may be assigned to the process's permitted capability set during an *exec()*. The corresponding *file inheritable* set is masked (ANDed) against the process inherited capability set to determine the capabilities that are actually added to the process's permitted capability set during an *exec()*.

> There is a further, philosophical reason for not simply preserving the process permitted capability set across an *exec()*. The idea of the capabilities system is that all privileges given to a process are granted or controlled by the file that the process execs. Although the process inheritable set specifies capabilities that are passed across an *exec()*, these capabilities are masked by the file inheritable set.

39.3.6 Assigning and Viewing File Capabilities from the Shell

The *setcap(8)* and *getcap(8)* commands, contained in the *libcap* package described in Section 39.7, can be used to manipulate file capabilities sets. We demonstrate the use of these commands with a short example using the standard *date(1)* program. (This program is an example of a capability-dumb application according to the definition in Section 39.3.4.) When run with privilege, *date(1)* can be used to change the system time. The *date* program is not set-user-ID-*root*, so normally the only way to run it with privilege is to become the superuser.

We begin by displaying the current system time, and then try to change the time as an unprivileged user:

```
$ date
Tue Dec 28 15:54:08 CET 2010
$ date -s '2018-02-01 21:39'
date: cannot set date: Operation not permitted
Thu Feb  1 21:39:00 CET 2018
```

Above, we see that the *date* command failed to change the system time, but nevertheless displayed its argument in the standard format.

Next, we become the superuser, which allows us to successfully change the system time:

```
$ sudo date -s '2018-02-01 21:39'
root's password:
Thu Feb  1 21:39:00 CET 2018
$ date
Thu Feb  1 21:39:02 CET 2018
```

We now make a copy of the *date* program and assign it the capability that it needs:

```
$ whereis -b date                            Find location of date binary
date: /bin/date
$ cp /bin/date .
$ sudo setcap "cap_sys_time=pe" date
root's password:
$ getcap date
date = cap_sys_time+ep
```

The *setcap* command shown above assigns the `CAP_SYS_TIME` capability to the permitted (*p*) and effective (*e*) capability sets of the executable file. We then used the *getcap* command to verify the capabilities assigned to the file. (The syntax used by *setcap* and *getcap* for representing capability sets is described in the *cap_from_text(3)* manual page provided in the *libcap* package.)

The file capabilities of our copy of the *date* program allow the program to be used by unprivileged users to set the system time:

```
$ ./date -s '2010-12-28 15:55'
Tue Dec 28 15:55:00 CET 2010
$ date
Tue Dec 28 15:55:02 CET 2010
```

39.4 The Modern Capabilities Implementation

A complete implementation of capabilities requires the following:

- For each privileged operation, the kernel should check whether the process has the relevant capability, rather than checking for an effective (or file system) user ID of 0.
- The kernel must provide system calls allowing a process's capabilities to be retrieved and modified.
- The kernel must support the notion of attaching capabilities to an executable file, so that the process gains the associated capabilities when that file is execed. This is analogous to the set-user-ID bit, but allows the independent specification of all capabilities on the executable file. In addition, the system must provide a set of programming interfaces and commands for setting and viewing the capabilities attached to an executable file.

Up to and including kernel 2.6.23, Linux met only the first two of these requirements. Since kernel 2.6.24, it is possible to attach capabilities to a file. Various other features were added in kernels 2.6.25 and 2.6.26 in order to complete the capabilities implementation.

For most of our discussion of capabilities, we'll focus on the modern implementation. In Section 39.10, we consider how the implementation differed before file capabilities were introduced. Furthermore, file capabilities are an optional kernel component in modern kernels, but for the main part of our discussion, we'll assume that this component is enabled. Later, we'll describe the differences that occur if file capabilities are not enabled. (In several respects, the behavior is similar to that of Linux in kernels before 2.6.24, where file capabilities were not implemented.)

In the following sections, we go into more detail on the Linux capabilities implementation.

39.5 Transformation of Process Capabilities During *exec()*

During an *exec()*, the kernel sets new capabilities for the process based on the process's current capabilities and the capability sets of the file being executed. The kernel calculates the new capabilities of the process using the following rules:

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

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

P'(inheritable) = P(inheritable)
```

In the above rules, *P* denotes the value of a capability set prior to the *exec()*, *P'* denotes the value of a capability set after the *exec()*, and *F* denotes a file capability set. The identifier *cap_bset* denotes the value of the capability bounding set. Note that *exec()* leaves the process inheritable capability set unchanged.

39.5.1 Capability Bounding Set

The capability bounding set is a security mechanism that is used to limit the capabilities that a process can gain during an *exec()*. This set is used as follows:

- During an *exec()*, the capability bounding set is ANDed with the file permitted capabilities to determine the permitted capabilities that are to be granted to the new program. In other words, an executable file's permitted capability set can't grant a permitted capability to a process if the capability is not in the bounding set.
- The capability bounding set is a limiting superset for the capabilities that can be added to the process's inheritable set. This means that, unless the capability is in the bounding set, a process can't add one of its permitted capabilities to its inheritable set and then—via the first of the capability transformation rules described above—have that capability preserved in its permitted set when it execs a file that has the capability in its inheritable set.

The capability bounding set is a per-process attribute that is inherited by a child created via *fork()*, and preserved across an *exec()*. On a kernel that supports file capabilities, *init* (the ancestor of all processes) starts with a capability bounding set that contains all capabilities.

If a process has the `CAP_SETPCAP` capability, then it can (irreversibly) remove capabilities from its bounding set using the *prctl()* `PR_CAPBSET_DROP` operation. (Dropping a capability from the bounding set doesn't affect the process permitted, effective, and inheritable capability sets.) A process can determine if a capability is in its bounding set using the *prctl()* `PR_CAPBSET_READ` operation.

> More precisely, the capability bounding set is a per-thread attribute. Starting with Linux 2.6.26, this attribute is displayed as the `CapBnd` field in the Linux-specific `/proc/`*PID*`/task/`*TID*`/status` file. The `/proc/`*PID*`/status` file shows the bounding set of a process's main thread.

39.5.2 Preserving *root* Semantics

In order to preserve the traditional semantics for the *root* user (i.e., *root* has all privileges) when executing a file, any capability sets associated with the file are ignored. Instead, for the purposes of the algorithm shown in Section 39.5, the file capability sets are notionally defined as follows during an *exec()*:

- If a set-user-ID-*root* program is being execed, or the real or effective user ID of the process calling *exec()* is 0, then the file inheritable and permitted sets are defined to be all ones.
- If a set-user-ID-*root* program is being execed, or the effective user ID of the process calling *exec()* is 0, then the file effective bit is defined to be set.

Assuming that we are execing a set-user-ID-*root* program, these notional definitions of the file capability sets mean that the calculation of the process's new permitted and effective capability sets in Section 39.5 simplifies to the following:

```
P'(permitted) = P(inheritable) | cap_bset
P'(effective) = P'(permitted)
```

39.6 Effect on Process Capabilities of Changing User IDs

To preserve compatibility with the traditional meanings for transitions between 0 and nonzero user IDs, the kernel does the following when changing process user IDs (using *setuid()*, and so on):

1. If the real user ID, effective user ID, or saved set-user-ID previously had the value 0 and, as a result of the changes to the user IDs, all three of these IDs have a nonzero value, then the permitted and effective capability sets are cleared (i.e., all capabilities are permanently dropped).
2. If the effective user ID is changed from 0 to a nonzero value, then the effective capability set is cleared (i.e., the effective capabilities are dropped, but those in the permitted set can be raised again).
3. If the effective user ID is changed from a nonzero value to 0, then the permitted capability set is copied into the effective capability set (i.e., all permitted capabilities become effective).

4. If the file-system user ID is changed from 0 to a nonzero value, then the following file-related capabilities are cleared from the effective capability 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). Conversely, if the file-system user ID is changed from a nonzero value to 0, then any of these capabilities that are enabled in the permitted set are enabled in the effective set. These manipulations are done to maintain the traditional semantics for manipulations of the Linux-specific file-system user ID.

39.7 Changing Process Capabilities Programmatically

A process can raise or drop capabilities from its capability sets using either the *capset()* system call or, preferably, the *libcap* API, which we describe below. Changes to process capabilities are subject to the following rules:

1. If the process doesn't have the CAP_SETPCAP capability in its effective set, then the new *inheritable* set must be a subset of the combination of the existing inheritable and permitted sets.
2. 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. In other words, a process can't grant itself permitted capabilities that it doesn't have. Put another way, a capability dropped from the permitted set can't be reacquired.
4. The new *effective* set is allowed to contain only capabilities that are also in the new permitted set.

The *libcap* API

Up to this point, we have deliberately not shown the prototype of the *capset()* system call, or its counterpart *capget()*, which retrieves a process's capabilities. This is because the use of these system calls should be avoided. Instead, the functions in the *libcap* library should be employed. These functions provide an interface that conforms with the withdrawn draft POSIX 1003.1e standard, along with some Linux extensions.

For reasons of space, we don't describe the *libcap* API in detail. As an overview, we note that programs employing these functions typically carry out the following steps:

1. Use the *cap_get_proc()* function to retrieve a copy of the process's current capability sets from the kernel and place it in a structure that the function allocates in user space. (Alternatively, we may use the *cap_init()* function to create a new, empty capability set structure.) In the *libcap* API, the *cap_t* data type is a pointer used to refer to such structures.
2. Use the *cap_set_flag()* function to update the user-space structure to raise (CAP_SET) and drop (CAP_CLEAR) capabilities from the permitted, effective, and inheritable sets stored in the user-space structure retrieved in the previous step.

3. Use the *cap_set_proc()* function to pass the user-space structure back to the kernel in order to change the process's capabilities.
4. Use the *cap_free()* function to free the structure that was allocated by the *libcap* API in the first step.

> At the time of writing, work is in progress on *libcap-ng*, a new, improved capabilities library API. Details can be found at *http://freshmeat.net/projects/libcap-ng*.

Example program

In Listing 8-2, on page 164, we presented a program that authenticates a username plus password against the standard password database. We noted that the program requires privilege in order to read the shadow password file, which is protected to prevent reading by users other than *root* or members of the *shadow* group. The traditional way of providing this program with the privileges that it requires would be to run it under a *root* login or to make it a set-user-ID-*root* program. We now present a modified version of this program that employs capabilities and the *libcap* API.

In order to read the shadow password file as a normal user, we need to bypass the standard file permission checks. Scanning the capabilities listed in Table 39-1, we see that the appropriate capability is `CAP_DAC_READ_SEARCH`. Our modified version of the password authentication program is shown in Listing 39-1. This program uses the *libcap* API to raise `CAP_DAC_READ_SEARCH` in its effective capability set just before accessing the shadow password file, and then drops the capability again immediately after this access. In order for an unprivileged user to employ the program, we must set this capability in the file permitted capability set, as shown in the following shell session:

```
$ sudo setcap "cap_dac_read_search=p" check_password_caps
root's password:
$ getcap check_password_caps
check_password_caps = cap_dac_read_search+p
$ ./check_password_caps
Username: mtk
Password:
Successfully authenticated: UID=1000
```

Listing 39-1: A capability-aware program that authenticates a user

cap/check_password_caps.c

```
#define _BSD_SOURCE        /* Get getpass() declaration from <unistd.h> */
#define _XOPEN_SOURCE      /* Get crypt() declaration from <unistd.h> */
#include <sys/capability.h>
#include <unistd.h>
#include <limits.h>
#include <pwd.h>
#include <shadow.h>
#include "tlpi_hdr.h"

/* Change setting of capability in caller's effective capabilities */

static int
modifyCap(int capability, int setting)
```

```
{
    cap_t caps;
    cap_value_t capList[1];

    /* Retrieve caller's current capabilities */

    caps = cap_get_proc();
    if (caps == NULL)
        return -1;

    /* Change setting of 'capability' in the effective set of 'caps'. The
       third argument, 1, is the number of items in the array 'capList'. */

    capList[0] = capability;
    if (cap_set_flag(caps, CAP_EFFECTIVE, 1, capList, setting) == -1) {
        cap_free(caps);
        return -1;
    }

    /* Push modified capability sets back to kernel, to change
       caller's capabilities */

    if (cap_set_proc(caps) == -1) {
        cap_free(caps);
        return -1;
    }

    /* Free the structure that was allocated by libcap */

    if (cap_free(caps) == -1)
        return -1;

    return 0;
}

static int              /* Raise capability in caller's effective set */
raiseCap(int capability)
{
    return modifyCap(capability, CAP_SET);
}

/* An analogous dropCap() (unneeded in this program), could be
   defined as: modifyCap(capability, CAP_CLEAR); */

static int              /* Drop all capabilities from all sets */
dropAllCaps(void)
{
    cap_t empty;
    int s;

    empty = cap_init();
    if (empty == NULL)
        return -1;

    s = cap_set_proc(empty);
    if (cap_free(empty) == -1)
        return -1;

    return s;
}

int
main(int argc, char *argv[])
{
    char *username, *password, *encrypted, *p;
    struct passwd *pwd;
    struct spwd *spwd;
    Boolean authOk;
    size_t len;
    long lnmax;

    lnmax = sysconf(_SC_LOGIN_NAME_MAX);
    if (lnmax == -1)                        /* If limit is indeterminate */
        lnmax = 256;                        /* make a guess */

    username = malloc(lnmax);
    if (username == NULL)
        errExit("malloc");

    printf("Username: ");
    fflush(stdout);
    if (fgets(username, lnmax, stdin) == NULL)
        exit(EXIT_FAILURE);                 /* Exit on EOF */

    len = strlen(username);
    if (username[len - 1] == '\n')
        username[len - 1] = '\0';           /* Remove trailing '\n' */

    pwd = getpwnam(username);
    if (pwd == NULL)
        fatal("couldn't get password record");

    /* Only raise CAP_DAC_READ_SEARCH for as long as we need it */

    if (raiseCap(CAP_DAC_READ_SEARCH) == -1)
        fatal("raiseCap() failed");

    spwd = getspnam(username);
    if (spwd == NULL && errno == EACCES)
        fatal("no permission to read shadow password file");

    /* At this point, we won't need any more capabilities,
       so drop all capabilities from all sets */

    if (dropAllCaps() == -1)
        fatal("dropAllCaps() failed");

    if (spwd != NULL)               /* If there is a shadow password record */
        pwd->pw_passwd = spwd->sp_pwdp;     /* Use the shadow password */
    password = getpass("Password: ");

    /* Encrypt password and erase cleartext version immediately */

    encrypted = crypt(password, pwd->pw_passwd);
    for (p = password; *p != '\0'; )
        *p++ = '\0';

    if (encrypted == NULL)
        errExit("crypt");

    authOk = strcmp(encrypted, pwd->pw_passwd) == 0;
    if (!authOk) {
        printf("Incorrect password\n");
        exit(EXIT_FAILURE);
    }

    printf("Successfully authenticated: UID=%ld\n", (long) pwd->pw_uid);

    /* Now do authenticated work... */

    exit(EXIT_SUCCESS);
}
```

cap/check_password_caps.c

39.8 Creating Capabilities-Only Environments

In the preceding pages, we have described various ways in which a process with the user ID 0 (*root*) is treated specially with respect to capabilities:

- When a process with one or more user IDs that equal 0 sets all of its user IDs to nonzero values, its permitted and effective capability sets are cleared. (See Section 39.6.)
- When a process with an effective user ID of 0 changes that user ID to a nonzero value, it loses its effective capabilities. When the reverse change is made, the permitted capability set is copied to the effective set. A similar procedure is followed for a subset of capabilities when the process's file-system user ID is switched between 0 and nonzero values. (See Section 39.6.)
- If a process with real or effective user ID of *root* execs a program, or any process execs a set-user-ID-*root* program, then the file inheritable and permitted sets are notionally defined to be all ones. If the process's effective user ID is 0, or it is execing a set-user-ID-*root* program, then the file effective bit is notionally defined to be 1. (See Section 39.5.2.) In the usual cases (i.e., both the real and effective user ID are *root*, or a set-user-ID-*root* program is being execed), this means the process gets all capabilities in its permitted and effective sets.

In a fully capability-based system, the kernel would not need to perform any of these special treatments of *root*. There would be no set-user-ID-*root* programs, and file capabilities would be used to grant just the minimum capabilities that a program requires.

Since existing applications aren't engineered to make use of the file-capabilities infrastructure, the kernel must maintain the traditional handling of processes with the user ID 0. Nevertheless, we may want an application to run in a purely capability-based environment in which *root* gets none of the special treatments described above. Starting with kernel 2.6.26, and if file capabilities are enabled in the kernel, Linux provides the *securebits* mechanism, which controls a set of per-process flags that enable or disable each of the three special treatments for *root*. (To be precise, the *securebits* flags are actually a per-thread attribute.)

The *securebits* mechanism controls the flags shown in Table 39-2. The flags exist as related pairs of a *base* flag and a corresponding *locked* flag. Each of the base flags controls one of the special treatments of *root* described above. Setting the corresponding locked flag is a one-time operation that prevents further changes to the associated base flag—once set, the locked flag can't be unset.

Table 39-2: The *securebits* flags

Flag	Meaning if set
`SECBIT_KEEP_CAPS`	Don't drop permitted capabilities when a process with one or more 0 user IDs sets all of its user IDs to nonzero values. This flag has an effect only if `SECBIT_NO_SETUID_FIXUP` is not also set. This flag is cleared on an *exec()*.
`SECBIT_NO_SETUID_FIXUP`	Don't change capabilities when effective or file-system user IDs are switched between 0 and nonzero values.
`SECBIT_NOROOT`	If a process with a real or effective user ID of 0 does an *exec()*, or it execs a set-user-ID-*root* program, don't grant it capabilities (unless the executable has file capabilities).
`SECBIT_KEEP_CAPS_LOCKED`	Lock `SECBIT_KEEP_CAPS`.
`SECBIT_NO_SETUID_FIXUP_LOCKED`	Lock `SECBIT_NO_SETUID_FIXUP`.
`SECBIT_NOROOT_LOCKED`	Lock `SECBIT_NOROOT`.

The *securebits* flag settings are inherited in a child created by *fork()*. All of the flag settings are preserved during *exec()*, except `SECBIT_KEEP_CAPS`, which is cleared for historical compatibility with the `PR_SET_KEEPCAPS` setting, described below.

A process can retrieve the *securebits* flags using the *prctl()* `PR_GET_SECUREBITS` operation. If a process has the `CAP_SETPCAP` capability, it can modify the *securebits* flags using the *prctl()* `PR_SET_SECUREBITS` operations. A purely capability-based application can irreversibly disable special treatment of *root* for the calling process and all of its descendants using the following call:

```
if (prctl(PR_SET_SECUREBITS,
        /* SECBIT_KEEP_CAPS off */
        SECBIT_NO_SETUID_FIXUP | SECBIT_NO_SETUID_FIXUP_LOCKED |
        SECBIT_NOROOT | SECBIT_NOROOT_LOCKED)
      == -1)
    errExit("prctl");
```

After this call, the only way in which this process and its descendants can obtain capabilities is by executing programs that have file capabilities.

SECBIT_KEEP_CAPS and the *prctl()* PR_SET_KEEPCAPS operation

The SECBIT_KEEP_CAPS flag prevents capabilities from being dropped when a process with one or more user IDs with the value 0 sets all of its user IDs to nonzero values. Roughly speaking, SECBIT_KEEP_CAPS provides half of the functionality provided by SECBIT_NO_SETUID_FIXUP. (As noted in Table 39-2, SECBIT_KEEP_CAPS has an effect only if SECBIT_NO_SETUID_FIXUP is not set.) This flag exists to provide a *securebits* flag that mirrors the older *prctl()* PR_SET_KEEPCAPS operation, which controls the same attribute. (The one difference between the two mechanisms is that a process doesn't need the CAP_SETPCAP capability to employ the *prctl()* PR_SET_KEEPCAPS operation.)

> Earlier, we noted that all of the *securebits* flags are preserved during an *exec()*, except SECBIT_KEEP_CAPS. The setting of the SECBIT_KEEP_CAPS bit was made the converse of the other *securebits* settings in order to maintain consistency with the treatment of the attribute set by the *prctl()* PR_SET_KEEPCAPS operation.

The *prctl()* PR_SET_KEEPCAPS operation is designed for use by set-user-ID-*root* programs running on older kernels that don't support file capabilities. Such programs can still improve their security by programmatically dropping and raising capabilities as required (refer to Section 39.10).

However, even if such a set-user-ID-*root* program drops all capabilities except those that it requires, it still maintains two important privileges: the ability to access files owned by *root* and the ability to regain capabilities by execing a program (Section 39.5.2). The only way of permanently dropping these privileges is to set all of the process's user IDs to nonzero values. But doing that normally results in the clearing of the permitted and effective capability sets (see the four points in Section 39.6 concerning the effect of user ID changes on capabilities). This defeats the purpose, which is to permanently drop user ID 0, while maintaining some capabilities. To allow this possibility, the *prctl()* PR_SET_KEEPCAPS operation can be used to set the process attribute that prevents the permitted capability set from being cleared when all user IDs are changed to a nonzero value. (The process's effective capability set is always cleared in this case, regardless of the setting of the "keep capabilities" attribute.)

39.9 Discovering the Capabilities Required by a Program

Suppose we have a program that is unaware of capabilities and that is provided only in binary form, or we have a program whose source code is too large for us to easily read to determine which capabilities might be required to run it. If the program requires privileges, but shouldn't be a set-user-ID-*root* program, then how can we determine the permitted capabilities to assign to the executable file with *setcap(8)*? There are two ways to answer this question:

- Use *strace(1)* (Appendix A) to see which system call fails with the error EPERM, the error used to indicate the lack of a required capability. By consulting the system call's manual page or the kernel source code, we can then deduce what capability is required. This approach isn't perfect, because an EPERM error can occasionally be generated for other reasons, some of which may have nothing to do with the capability requirements for the program. Furthermore, programs may legitimately make a system call that requires privilege, and then change their behavior after determining that they don't have privilege for a particular operation. It can sometimes be difficult to distinguish such "false positives" when trying to determine the capabilities that an executable really does need.

- Use a kernel probe to produce monitoring output when the kernel is asked to perform capability checks. An example of how to do this is provided in [Hallyn, 2007], an article written by one of the developers of file capabilities. For each request to check a capability, the probe shown in the article logs the kernel function that was called, the capability that was requested, and the name of the requesting program. Although this approach requires more work than the use of *strace(1)*, it can also help us more accurately determine the capabilities that a program requires.

39.10 Older Kernels and Systems Without File Capabilities

In this section, we describe various differences in the implementation of capabilities in older kernels. We also describe the differences that occur on kernels where file capabilities are not supported. There are two scenarios where Linux doesn't support file capabilities:

- Before Linux 2.6.24, file capabilities were not implemented.
- Since Linux 2.6.24, file capabilities can be disabled if the kernel is built without the `CONFIG_SECURITY_FILE_CAPABILITIES` option.

> Although Linux introduced capabilities and allowed them to be attached to processes starting with kernel 2.2, the implementation of file capabilities appeared only several years later. The reasons that file capabilities remained unimplemented for so long were matters of policy, rather than technical difficulties. (Extended attributes, described in Chapter 16, which are used to implement file capabilities, had been available since kernel 2.6.) The weight of opinion among kernel developers was that requiring system administrators to set and monitor different sets of capabilities—some of whose consequences are subtle but far-reaching—for each privileged program would create an unmanageably complex administration task. By contrast, system administrators are familiar with the existing UNIX privilege model, know to treat set-user-ID programs with due caution, and can locate the set-user-ID and set-group-ID programs on a system using simple *find* commands. Nevertheless, the developers of file capabilities made the case that file capabilities could be made administratively workable, and eventually provided a convincing enough argument that file capabilities were integrated into the kernel.

The `CAP_SETPCAP` capability

On kernels that don't support file capabilities (i.e., any kernel before 2.6.24, and kernels since 2.6.24 with file capabilities disabled), the semantics of the `CAP_SETPCAP` capability are different. Subject to rules that are analogous to those described in Section 39.7, a process that has the `CAP_SETPCAP` capability in its effective set can theoretically change the capabilities of processes other than itself. Changes can be made to the capabilities of another process, all of the members of a specified process group, or all processes on the system except *init* and the caller itself. The final case excludes *init* because it is fundamental to the operation of the system. It also excludes the caller because the caller may be attempting to remove capabilities from every other process on the system, and we don't want to remove the capabilities from the calling process itself.

However, changing the capabilities of other processes is only a theoretical possibility. On older kernels, and on modern kernels where support for file capabilities is disabled, the capability bounding set (discussed next) always masks out the `CAP_SETPCAP` capability.

The capability bounding set

Since Linux 2.6.25, the capability bounding set is a per-process attribute. However, on older kernels, the capability bounding set is a system-wide attribute that affects all processes on the system. The system-wide capability bounding set is initialized so that it always masks out `CAP_SETPCAP` (described above).

> On kernels after 2.6.25, removing capabilities from the per-process bounding set is supported only if file capabilities are enabled in the kernel. In that case, *init*, the ancestor of all processes, starts with a bounding set containing all capabilities, and a copy of that bounding set is inherited by other processes created on the system. If file capabilities are disabled, then, because of the differences in the semantics of `CAP_SETPCAP` described above, *init* starts with a bounding set that contains all capabilities except `CAP_SETPCAP`.

There is one further change in the semantics of the capability bounding set in Linux 2.6.25. As noted earlier (Section 39.5.1), on Linux 2.6.25 and later, the per-process capability bounding set acts as a limiting superset for the capabilities that can be added to the process's inheritable set. In Linux 2.6.24 and earlier, the system-wide capability bounding set doesn't have this masking effect. (It is not needed, because these kernels don't support file capabilities.)

The system-wide capability bounding set is accessible via the Linux-specific `/proc/sys/kernel/cap-bound` file. A process must have the `CAP_SYS_MODULE` capability to be able to change the contents of `cap-bound`. However, only the *init* process can turn bits on in this mask; other privileged processes can only turn bits off. The upshot of these limitations is that on a system where file capabilities are not supported, we can never give the `CAP_SETPCAP` capability to a process. This is reasonable, since that capability can be used to subvert the entire kernel privilege-checking system. (In the unlikely case that we want to change this limitation, we must either load a kernel module that changes the value in the set, modify the source code of the *init* program, or change the initialization of the capability bounding set in the kernel source code and perform a kernel rebuild.)

> Confusingly, although it is a bit mask, the value in the system-wide `cap-bound` file is displayed as a signed decimal number. For example, the initial value of this file is –257. This is the two's complement interpretation of the bit mask with all bits except *(1 << 8)* turned on (i.e., in binary, 11111111 11111111 11111110 11111111); `CAP_SETPCAP` has the value 8.

Using capabilities within a program on a system without file capabilities

Even on a system that doesn't support file capabilities, we can nevertheless employ capabilities to improve the security of a program. We do this as follows:

1. Run the program in a process with an effective user ID of 0 (typically a set-user-ID-*root* program). Such a process is granted all capabilities (except CAP_SETPCAP, as noted earlier) in its permitted and effective sets.
2. On program startup, use the *libcap* API to drop all capabilities from the effective set, and drop all capabilities except those that we may later need from the permitted set.
3. Set the SECBIT_KEEP_CAPS flag (or use the *prctl()* PR_SET_KEEPCAPS operation to achieve the same result), so that the next step does not drop capabilities.
4. Set all user IDs to nonzero values, to prevent the process from accessing files owned by *root* or gaining capabilities by doing an *exec()*.

   > We could replace the two preceding steps by a single step that sets the SECBIT_NOROOT flag, if we want to prevent the process from regaining privileges on an *exec()*, but must allow it to access files owned by *root*. (Of course, allowing access to files owned by *root* leaves open the risk of some security vulnerability.)

5. During the rest of the program's lifetime, use the *libcap* API to raise and drop the remaining permitted capabilities from the effective set as needed in order to perform privileged tasks.

Some applications built for Linux kernels before version 2.6.24 employed this approach.

> Among the kernel developers who argued against the implementation of capabilities for executable files, one of the perceived virtues of the approach described in the main text was that the developer of an application knows which capabilities an executable requires. By contrast, a system administrator may not be able to easily determine this information.

39.11 Summary

The Linux capabilities scheme divides privileged operations into distinct categories, and allows a process to be granted some capabilities, while being denied others. This scheme represents an improvement over the traditional all-or-nothing privilege mechanism, whereby a process has either privileges to perform all operations (user ID 0) or no privileges (nonzero user ID). Since kernel 2.6.24, Linux supports attaching capabilities to files, so that a process can gain selected capabilities by execing a program.

39.12 Exercise

39-1. Modify the program in Listing 35-2 (`sched_set.c`, on page 743) to use file capabilities, so that it can be used by an unprivileged user.