Linux-Privs

Andrew G. Morgan, morgan@parc.power.net DRAFT v0.10 1997/4/21

This is the specification document for Linux-Privs. It is a "standard" describing the kernel component to a secure system based around a Linux Kernel and is designed to be as compatible with POSIX.1e as is possible without open access to the actual specifications.

Contents

1 Introduction **6**

2 Capabilities **7**

2.1 POSIX capabilities 7

2.1.1 CAP_CHOWN 7

2.1.2 CAP_DAC_OVERRIDE 7

2.1.3 CAP_DAC_READ_SEARCH 7

2.1.4 CAP_FOWNER 8

2.1.5 CAP_FSETID 8

2.1.6 CAP_KILL 8

2.1.7 CAP_LINK_DIR 8

2.1.8 CAP_SETFCAP 8

2.1.9 CAP_SETGID 8

2.1.10 CAP_SETUID 8

2.1.11 CAP_SIGMASK 8

2.1.12 CAP_MAC_DOWNGRADE 9

2.1.13 CAP_MAC_READ 9

2.1.14 CAP_MAC_RELABEL_SUBJ 9

2.1.15 CAP_MAC_UPGRADE 9

2.1.16 CAP_MAC_WRITE 9

2.1.17 CAP_INF_NOFLOAT_OBJ 9

2.1.18 CAP_INF_NOFLOAT_SUBJ 9

2.1.19 CAP_INF_RELABEL_OBJ 9

2.1.20 CAP_INF_RELABEL_SUBJ 10

2.1.21 CAP_AUDIT_CONTROL . . . 10
2.1.22 CAP_AUDIT_WRITE . . . 10
2.2 Linux specific capabilities . . . 10
2.2.1 CAP_LINUX_IMMUTABLE . . . 10
2.2.2 CAP_LINUX_KERNELD . . . 10
2.2.3 CAP_LINUX_INSMOD . . . 10
2.2.4 CAP_LINUX_RMMOD . . . 10
2.2.5 CAP_LINUX_RAWIO . . . 10
2.2.6 CAP_LINUX_ATTENTION . . . 11
2.2.7 CAP_LINUX_RANDOM . . . 11
2.3 Other capabilities . . . 11
2.3.1 CAP_NET_BIND_SERVICE . . . 11
2.3.2 CAP_NET_BROADCAST . . . 11
2.3.3 CAP_NET_DEBUG . . . 11
2.3.4 CAP_NET_FIREWALL . . . 11
2.3.5 CAP_NET_IFCONFIG . . . 11
2.3.6 CAP_NET_PACKET . . . 11
2.3.7 CAP_NET_RAW . . . 11
2.3.8 CAP_NET_ROUTE . . . 12
2.3.9 CAP_NET_SETID . . . 12
2.3.10 CAP_IPC_LOCK . . . 12
2.3.11 CAP_IPC_OWNER . . . 12
2.3.12 CAP_SYS_CHROOT . . . 12
2.3.13 CAP_SYS_PTRACE . . . 12
2.3.14 CAP_SYS_ACCOUNT . . . 12
2.3.15 CAP_SYS_ADMIN . . . 12
2.3.16 CAP_SYS_BOOT . . . 12
2.3.17 CAP_SYS_DEVICES . . . 12
2.3.18 CAP_SYS_NICE . . . 13
2.3.19 CAP_SYS_RESOURCE . . . 13
2.3.20 CAP_SYS_TIME . . . 13
2.3.21 CAP_SYS_TTY_CONFIG . . . 13

2.3.22 CAP_SYS_QUOTA . . . 13

3 Capability sets **14**

3.1 Sets of capabilities . . . 14

3.1.1 The capability set type . . . 14

3.1.2 Capability set macros . . . 14

3.2 Filesystem capabilitiy sets . . . 15

3.2.1 Inheritable (fI) . . . 15

3.2.2 Permitted (fP) . . . 15

3.2.3 Effective (fE) . . . 15

3.3 Process/task capabilities . . . 16

3.3.1 Effective (pE) . . . 16

3.3.2 Inheritable (pI) . . . 16

3.3.3 Permitted (pP) . . . 16

4 Access Control Lists (ACL) **18**

5 Mandatory Access Control (MAC) **19**

6 Information Labels (IL) **20**

7 Sensitivity Labels (SL) **21**

8 Integrity checking; system recovery **22**

9 Auditing **23**

9.1 First some kernel infrastructure: . . . 23

9.2 What audit records should contain? . . . 24

9.3 Next some Policy (when/what should we audit): . . . 24

9.4 User level auditing (POSIX): . . . 25

9.5 Internal kernel auditing facility . . . 27

9.6 Preserving the audit log . . . 27

9.7 Auditing policy . . . 27

10 Process/task credentials **28**

11 Acknowledgements **29**

12 Copyright/license for this document **30**

Chapter 1

Introduction

The Linux-Privs project is an effort to implement a POSIX.1e (formerly POSIX 6) security model under Linux.

At the heart of the changes to the Linux kernel from the historical model is the separation of identity and privilege. Historically, root (UID=0) was all powerful and other users (UID!=0) had power that was limited to that associated with their identity and their group memberships.

The Linux-Privs scheme is to implement a set of independent capabilities that can be given to any user. In point of fact, the capabilities are associated with applications and can only be used within the confines of the functionality of such applications. In this way individual capabilities can be restricted to a trusted set of applications. Typically, the user will have to authenticate himself to such an applciation before it will perform its privileged task.

This new scheme for system privilege lends itself well to restricting privileged access to the system and reduces the risk of intruders or poorly written applications running amok on the system.

Chapter 2

Capabilities

In this chapter we list the capabilities known to the Linux Kernel. Firstly, we list the POSIX defined capabilities, and then those specific to Linux.

2.1 POSIX capabilities

Here we list the POSIX capabilities honored by Linux.

2.1.1 CAP_CHOWN

The `#define` for the symbol `_POSIX_CHOWN_RESTRICTED`, indicates that this capability (`CAP_CHOWN`) is known.

This capability enables the current process to change the owner of a file. Generally, file ownership is not changeable by a user: it is implied by the user that creates it.

2.1.2 CAP_DAC_OVERRIDE

The `#define` for the symbol **_POSIX_ACL**, indicates that Access Control Lists (an implementation of Discretionary Access Control) are supported by the kernel and that the following capabilities is known: `CAP_DAC_OVERRIDE`.

This capability overrides all DAC restrictions regarding read and search on files and directories, including ACL's.

2.1.3 CAP_DAC_READ_SEARCH

Overrides all restrictions about allowed operations on files, where file owner ID must be equal to the user ID, except where CAP_FSETID is applicable. It doesn't override MAC and DAC restrictions.

2.1.4 CAP_FOWNER

Overrides the following restrictions that the effective user ID shall match the file owner ID when setting the S_ISUID and S_ISGID bits on that file; that the effective group ID (or one of the supplementary group IDs) shall match the file's group owner ID when setting the S_ISGID bit on that file; that the S_ISUID and S_ISGID bits are cleared on successful return from chown(2).

2.1.5 CAP_FSETID

Overrides the restriction that the real or effective user ID of a process sending a signal must match the real or effective user ID of the process receiving the signal.

2.1.6 CAP_KILL

Overrides the restriction that the real or effective user ID of a process sending a signal must match the real or effective user ID of the process receiving the signal.

2.1.7 CAP_LINK_DIR

Overrides the restriction that a process cannot create or delete a hard link to a directory. This shall not override MAC and DAC policies.

2.1.8 CAP_SETFCAP

Allows the (re)setting of a files capabilities.

2.1.9 CAP_SETGID

Allows setgid(2) manipulation.

2.1.10 CAP_SETUID

Allows setuid(2) manipulation

2.1.11 CAP_SIGMASK

Overrides the restriction that no process may block `SIGKILL` and `SIGSTOP`.

2.1.12 CAP_MAC_DOWNGRADE

This capability is available if `_POSIX_MAC` is `#define`'d. This capability allows a process to downgrade an object's information label.

2.1.13 CAP_MAC_READ

This capability is available if `_POSIX_MAC` is `#define`'d. Allows a process to override MAC read restrictions.

2.1.14 CAP_MAC_RELABEL_SUBJ

This capability is available if `_POSIX_MAC` is `#define`'d. Allows a process to modify its own label.

2.1.15 CAP_MAC_UPGRADE

This capability is available if `_POSIX_MAC` is `#define`'d. This capability allows a process to upgrade an object's information label.

2.1.16 CAP_MAC_WRITE

This capability is available if `_POSIX_MAC` is `#define`'d. This capability overrides the MAC restrictions on writes.

2.1.17 CAP_INF_NOFLOAT_OBJ

This capability is available if `_POSIX_INF` is `#define`'d. This capability prevents a process' information label from floating during writes.

2.1.18 CAP_INF_NOFLOAT_SUBJ

This capability is available if `_POSIX_INF` is `#define`'d. This capability prevents the process' information label from floating during reads or executes.

2.1.19 CAP_INF_RELABEL_OBJ

This capability is available if `_POSIX_INF` is `#define`'d. This capability allows a process to change an object's information label.

2.1.20 CAP_INF_RELABEL_SUBJ

This capability is available if `_POSIX_INF` is `#define`'d. This capability allows a process to modify its own information label in violation of the overriding policy.

2.1.21 CAP_AUDIT_CONTROL

This capability is available if `_POSIX_AUD` is `#define`'d. This capability allows a process to modify the audit control parameters.

2.1.22 CAP_AUDIT_WRITE

This capability is available if `_POSIX_AUD` is `#define`'d. This capability allows a process to write data to the audit trail.

2.2 Linux specific capabilities

This section lists additional capabilities that are specific to Linux or not covered by the POSIX capability definitions.

2.2.1 CAP_LINUX_IMMUTABLE

Allow modification of `S_IMMUTABLE` and `S_APPEND` file attributes.

2.2.2 CAP_LINUX_KERNELD

Permission to act as kerneld.

2.2.3 CAP_LINUX_INSMOD

Allow installation of kernel modules.

2.2.4 CAP_LINUX_RMMOD

Allow removal of kernel modules.

2.2.5 CAP_LINUX_RAWIO

Allow ioperm/iopl access.

2.2.6 CAP_LINUX_ATTENTION

Allow configuration of the secure attention key.

2.2.7 CAP_LINUX_RANDOM

Allow administration of the random device.

2.3 Other capabilities

This sction lists those capabilities commonly found on other systems besides Linux, but which are not specified by POSIX.

2.3.1 CAP_NET_BIND_SERVICE

Allows binding to TCP/UDP sockets below 1024.

2.3.2 CAP_NET_BROADCAST

Allow broadcasting.

2.3.3 CAP_NET_DEBUG

Allow setting debug option on sockets.

2.3.4 CAP_NET_FIREWALL

Allow configuring of firewall stuff.

2.3.5 CAP_NET_IFCONFIG

Allow interface configuration.

2.3.6 CAP_NET_PACKET

Allow use of PACKET sockets.

2.3.7 CAP_NET_RAW

Allow use of RAW sockets.

2.3.8 CAP_NET_ROUTE

Allow modification of routing tables.

2.3.9 CAP_NET_SETID

CAP.FIXME: what is this about?.

2.3.10 CAP_IPC_LOCK

Allow locking of segments in memory.

2.3.11 CAP_IPC_OWNER

Override IPC ownership checks.

2.3.12 CAP_SYS_CHROOT

Allow use of chroot().

2.3.13 CAP_SYS_PTRACE

Allow ptrace() of any process.

2.3.14 CAP_SYS_ACCOUNT

Allow configuration of process accounting.

2.3.15 CAP_SYS_ADMIN

System Admin functions: mount et al.

2.3.16 CAP_SYS_BOOT

Allow use of reboot().

2.3.17 CAP_SYS_DEVICES

Allow device administration.

2.3.18 CAP_SYS_NICE

Allow use of renice() on others, and raising of priority.

2.3.19 CAP_SYS_RESOURCE

Override resource limits.

2.3.20 CAP_SYS_TIME

Allow manipulation of system clock.

2.3.21 CAP_SYS_TTY_CONFIG

Allow configuration of tty devices.

2.3.22 CAP_SYS_QUOTA

Allow examination and configuration of disk quotas.

Chapter 3

Capability sets

In this chapter we introduce the concept of sets of capabilities. We also discuss the way in which such sets are maintained by the kernel and their association with system objects.

3.1 Sets of capabilities

Capabilities are independent and an application may have any number of them raised. The total selection of raised capabilities constitutes a capability set.

3.1.1 The capability set type

The kernel represents a capability set with the structure `struct __cap_s`. and may be thought of as a bitmap. The number of capability bits is currently 128.

There is a special, empty, capabiltiy set. This set has no capabilities raised.

3.1.2 Capability set macros

To manipulate capabilities, a kernel task uses one of the following three macros:

_cap_raise(cap_value_t capability)

This macro raises a capability in a capability set. It is used (with respect to a capability "struct __cap_s c") in the following manner:

```
c._cap_raise(CAP_AUDIT_CONTROL);
```

Note, the macro is used in place of a member of the `__cap_s`.

_cap_lower(cap_value_t capability);

This macro lowers a capability in a capability set. It is used (with respect to a capability "struct __cap_s c") in the following manner:

```
c._cap_lower(CAP_AUDIT_CONTROL);
```

Note, the macro is used in place of a member of the `__cap_s`.

_cap_raised(cap_value_t capability);

This macro returns *true*(!=0) if the capability is raised, and *false*(=0) if the capability was lowered. It is used in the following manner:

```
if (c._cap_raised(CAP_AUDIT_CONTROL)) {

   /* .... action in case that capability is raised .... */

}
```

Note, this macro is used as if it were a member of the capability structure.

3.2 Filesystem capabilitiy sets

Files on the system have the following capability sets.

3.2.1 Inheritable (fI)

This is the set of capabilities that the program is permitted to inherit. In this way, the program can be configured to be safe for operation with a fixed number of capabilities. (This has the feature of eliminating the total colapse of system security in the face of an insecurely written game.)

3.2.2 Permitted (fP)

This is the set of capabilities that are explicitly raised when the new process is `exec()`'d.

3.2.3 Effective (fE)

This is a single valued capability set (0 or 0) that is required to be set if the process (after an exec) is to have all of its capabilities automatically raised. It is intended to be used with programs that were written before the concept of capabilities was introduced. Newly written programs should raise their own capabilities manually.

3.3 Process/task capabilities

Processes (also know as kernel-tasks) have the following capability sets. They are copied unchanged when the process performs a `fork()` (or more generally, a `clone()`) function call.

Capability sets for a process are always affected by a call to `exec()`. Additionally, user level programs can manipulate their own capabilities with system calls to `sys_[gs]etproccap()`.

Capabilities are grouped into sets and come in three forms: Effective; Inheritable; and Permitted.

3.3.1 Effective (pE)

This is the set of capabilities that are currently *raised*, or in effect.

This is set upon `exec()` to

```
pE' = pP' & fE.
```

In other words, the new effective set is the new permitted set masked with the `exec()`uted file's effective set..

Capabilities may be raised in the Effective set only if they are already raised in the Permitted set. Capabilities not in the Effective set are "lowered", or are inactive.

3.3.2 Inheritable (pI)

This is the set of capabilities that can be passed through an `exec()`. These capabilities do not have to be raised in the predecessor to be passed, just present in the old process sets.

Following an exec(), the Inheritable set becomes

```
pI' = pI.
```

Or in other words, the Inheritable set is passed unchanged thrhough and `exec()`..

3.3.3 Permitted (pP)

These are the capabilities that are potentially available to the current task.

Following an `exec()`, the process' Permitted set is constructed from the following combination of the former, and invoked, file's capability sets:

```
pP' = fP | ( fI & pI ).
```

Or in other words, the process' permitted set becomes the combination of the permitted set of the `exec()`'d file and those inheritable capabilities of the `exec()`ing file that are also inheritable by the file.

Turning off a capability set in the Permitted set does not affect its status in the Inheritable set. The following is apparently a quote: (draft 13 of POSIX .1e) POSIX B.15.1.9

"The corresponding inheritable flag is not affected, so that privileges can be conditionally transmitted along a process chain whose intermediate processes may themselves have no privileges" (N.B. At this stage in the development of the POSIX standard, capabilities were termed privileges.) So if you want to turn off the capabilities in the Inheritable set, you should explicitly turn it off, not just turn it off in the Permitted set.

Chapter 4

Access Control Lists (ACL)

Chapter 5

Mandatory Access Control (MAC)

Chapter 6

Information Labels (IL)

Chapter 7

Sensitivity Labels (SL)

Chapter 8

Integrity checking; system recovery

Chapter 9

Auditing

An auditing facility is provided. Its design is intended to be simple but flexible, in such a way that it can be robustly implemented and will not impinge on the general responsiveness of the Linux kernel.

The following is a summary of the recent discussion on the linux-privs list. I am in the process of making a proposal that will be general enough to handle all of the following features:

9.1 First some kernel infrastructure:

- basic (atomic) auditing facility
  - two modes of access: blocking and non-blocking
- some sort of circular buffer arrangement with hooks for a module of some sort to store the audit data on some medium.
  - such a module will accept rules that tell it what to audit and what to discard.
  - perhaps we adopt the model of the firewalling rules.
- audit - session id for following login etc. process trees.
- minimal audit records include audit-id, uid and time plus raw numerical data (for off-line translation to readable text
- all audit records have the same prefix-header
  - Fixed Header prefix contains a version number (audit readers will know how to deal with historical formats).

9.2 What audit records should contain?

- Events to be audited are selected by event masks: union of system/user/file event masks.
- Proposed audit prefix header:
  - audit header version (audit generation) __u8
  - length of audit record (bytes) __u16
  - machine id: __u16
  - kind of event (e.g. system call code): __u8
  - reason for logging (bitmask: audited syscall, audited file, audited user, audited group) __u32
  - timestamp (when audited) __u32
  - audit id (set by init or login etc., for following process evolution) __u32
  - user id, group id, program id, ... __u32

9.3 Next some Policy (when/what should we audit):

- Classes of event
  - Group individual events into classes: file-access/system tweeking etc..
  - Audit primarily on class of event.
- Events that will always be audited (fixed events):
  - all actions relating to the audit sybsystem itself
  - all attempts to change the system date
  - all actions relating to group and user attributes
  - all definitions and deletions of MAC level names and level identifiers
  - chages of init states
  - all actions relating to lodable modules
- User level event classes:
  - some method for defining user-level event groups: administrator can select etc.
- Events that may be audited
  - file-access (open, connect, accept, close)
  - changing of file attributes (chmod, chattr, umask, create)
  - program execution (exec, fork, vfork, exit)
  - "exits" - administrator hook to deny requests/grant requests/modify system etc., in the light of the event.

9.4 User level auditing (POSIX):

- POSIX only defines a method for writing audit records.
- Audit records are variable length
- each record contains header
  - event type - indicates minimal information contained in record
  - time
  - result (success/failure)
  - audit - id
  - other stuff...(?)
- subject attributes
  - information about the calling process.
    * pid
- object structures
  - contain information on objects associated with the audited event:
    * files
    * processes
    * etc.
- event specific data: as directed by the event type.
- Functions:
  - int aud_write(int fildes, aur_rec_t ar)
    * special fildes = AUD_SYSTEM_LOG, to indicate that data is written to the system audit record.
- Policy: here we list audit event types (and the functions that they refer to)
  - AUD_AET_AUD_SWITCH
    * aud_switch()
  - AUD_AET_AUD_WRITE
    * aud_write()
  - AUD_AET_CHDIR
    * chdir()
  - AUD_AET_CHMOD

  * chmod()
- AUD_AET_CHOWN
  * chown()
- AUD_AET_CREAT
  * creat()
- AUD_AET_DUP
  * dup(), dup2()
- AUD_AET_EXEC
  * exec(), execl(), execlp(), execv(), execvp(), execle(), execve()
- AUD_AET_EXIT
  * _exit()
- AUD_AET_FORK
  * fork()
- AUD_AET_KILL
  * kill()
- AUD_AET_LINK
  * link()
- AUD_AET_MKDIR
  * mkdir()
- AUD_AET_MKFIFO
  * mkfifo()
- AUD_AET_OPEN
  * open(), opendir()
- AUD_AET_PIPE
  * pipe()
- AUD_AET_RENAME
  * rename()
- AUD_AET_RMDIR
  * rmdir()
- AUD_AET_SETGID
  * setgid()
- AUD_AET_SETUID
  * setuid()

  - AUD_AET_UNLINK
    * unlink()
  - AUD_AET_UTIME
    * utime()

- Define a minimus set of events that cannot be turned off: these preserve the integrety of the audit lof
- user and file bitmasks are provided to trace the behavior of the system with respect to specific files and users. These are OR'd together to determine if an event is to be logged.

9.5 Internal kernel auditing facility

Here we discuss the manner in which the kernel records audited events.

Any part of the kernel may audit an event.

Only those user-level processes with their auditing capability set may audit an event with the kernel.

Contingency is made for the possibility of the system log filling up.

9.6 Preserving the audit log

This section discusses the kernel hooks provided for making a hard-copy of the system log.

9.7 Auditing policy

This section describes the policies used to define what sorts of events are logged. It also introduces the preferred format for recording these audited events.

Chapter 10

Process/task credentials

In this section we cover the freedoms of a given process. These include items like resource limits.

Chapter 11

Acknowledgements

This document is edited by Andrew G. Morgan. Amendments and corrections should be emailed to <`linux-privs@mit.edu`>.

Contributors to the text of this document include: Hildo Biersma, Roland Buresund, Dave Dillow, Julie Haugh, David Holland, Alexander Kjeldaas, Andi Kleen, Darren Moffat, Ingo Molnar, Aleph One, Christos Ricudis, Ken Seefried, Theodore Ts'o, and Zefram (Andrew Main).

Chapter 12

Copyright/license for this document