Linux Capabilities: making them work

Serge E. Hallyn
IBM LTC
serue@us.ibm.com

Andrew G. Morgan
Google Inc.
agm@google.com

Abstract

Linux capabilities have been partially implemented for many years, and in their incomplete state have been nearly unusable. In light of recent kernel developments, including VFS support and per-process support for bounding-set and secure-bits, capabilities have finally come of age. In this paper we demonstrate, with examples, how capabilities enhance the security of the modern Linux system.

1 Introduction

Linux helps users manage their data, and a single Linux system simultaneously manages the data of multiple users. Within the system, a particular user's property is generally contained in *files* which are annotated with a numerical ownership user-identifier (UID). Linux also manages and abstracts the computer hardware, offering programs an environment in which to execute. Part of this abstraction enforces data ownership. In order to honor the ownership of such data, Linux adheres to context-specific rules limiting how programs can manipulate, via *system-calls*, a specific user's data (the context in this case being the value of attributes like the user's UID).

To start running programs on a Linux system, an applicant-user generally leverages a program (such as `login`, `sshd`, or `gdm`) to authenticate their identity to the system and create a working context for them to invoke other programs that access their data. Such *login* programs are exceptionally special, insofar as they have the ability to change the user context (set the *current* UID). Changing user context is clearly a special operation, since if it were not then programs run from the context of any user could trivially be leveraged to create a different user's context and manipulate data belonging to that other user. The special property of these applications is commonly known as *privilege*, and this paper concerns a newly completed mechanism for managing privilege within the Linux operating system.

Programs, in the context of authenticated users, can create data with access controls associated with them: create a file that anyone can read; create a file that only the creator can read or modify; etc. These forms of protection are known as Discretionary Access Control (DAC), and with more recent Linux extensions such as Access Control Lists (ACLs) can be quite elaborate [1]. The protection of such data is at the discretion of the owner of these files. Other mechanisms, such as Mandatory Access Control (MAC), enforce a system policy that restricts the ways in which users can share their data. Linux, via the Linux Security Module (LSM) [2] programming abstraction, natively supports simple MAC [3] and the more modern *type-enforcement* model [4, 5]. All of these mechanisms follow a tradition [6] of attempting to add real security to Linux and UNIX [7].

Managing a Linux system in the real world requires levels of reliability (continuity of service for multiple simultaneous users and uses) that must anticipate future problems: the need for data backups; configuration changes and upgrades to failing/obsolete hardware etc. There is also a recurrent need to work around and correct urgent issues: users accidentally instructing programs to delete files, users being removed from the system (and their data being archived or redistributed to other users); etc. These requirements, in addition to the need to *login* users (discussed above), lead to the fact that any system must provide the ability to override DAC and MAC security protections to "get things done." Viable systems need a *privilege* model.

The classic UNIX model for wielding privilege is to assign a special UID the right to do anything. Programs running in the context of this *super-user* are not bound by the normal DAC/MAC rules. They can read, modify, and change any user's data. In UNIX, UID=0 is the special context assigned to this administrative identity. To give this entity a more human touch, this user is also known as `root`. What this means for programs is that, when they run in the context of `root`'s UID, system-calls can do special (privileged) things. The converse of this is also significant: when they run in this context, programs can't help breaking the normal DAC/MAC rules, potentially causing unintended damage to the system. For example, executing: `rm -rf /`, as `root` can have spectacularly bad consequences to a system. However, running this command as a *normal user* (in this paper, we'll call such a user: `luser`) results in a prompt error and no negative effects. This `luser` doesn't have a right to delete anything in the base directory of the filesystem.

Unprivileged users, however, must always perform some tasks which do require privilege. For instance, our `luser` must be able to change his password. That being said, the system must prevent `luser` from being able to read or change passwords for other users. Users execute programs that act for them, and a program exists to change passwords: `passwd`. This program must be invoked from the context of the `luser` but operate with sufficient privilege to manipulate the shared-system's password file (`/etc/shadow`). To this end, Linux executable files can have a special attribute *setuid*-bit set, meaning that the program can execute with the *effective* UID (EUID[1]) of the user owning the program file. If the *setuid* `passwd` program file's owner is `root`,[2] then independent of the user context in which the program is launched, it will execute with the effective context of the super-user. That is, a program such as `passwd` will not be bound by any of the DAC/MAC rules that constrain other *regular* programs.

2 The Linux capability model

While the simple UNIX privilege mechanism has more or less sufficed for decades, it has long been observed that it has a significant shortcoming: that programs that require only some privilege must in fact run with full privilege. The dangers of such a lack of flexibility are well known, as they ensure that programming errors in privileged programs can be leveraged by hostile users to lead to full system compromise [8]. Such compromises can be mitigated through the use of MAC, but at some fundamental level any privileged access to the hardware underpinning an operating system can violate even MAC rules, and bogging MAC implementations down with details about `root` privilege separation only increases policy complexity. In the real world, administrative access to override normal access control mechanisms is a necessary feature.

Over the years, the proponents of a more secure UNIX [7] explored various alternatives to the concept of an all powerful `root` user. An aborted attempt was even made to unify these enhancements into a single standard [9]. The downward trajectory in the mid to late 1990's of the closed-source vendor-constrained rival commercial UNIX implementations mired, and eventually halted, the ratification of this *standard*. However, not entirely disconnected from this slowdown was the rapid and perhaps inevitable rise of Linux—a truly open (free) system in the original spirit of the UNIX tradition. These modern ideas of incremental enhancements to the UNIX security model have now found a home in Linux [1, 10, 3, 11].

The proposed privilege model [9] introduced a separation of root privilege into a set of *capabilities*. These capabilities break the super-user's privilege into a set of meaningfully separable privileges [7]. In Linux, for instance, the ability to switch UIDs is enabled by `CAP_SETUID` while the ability to change the ownership of an object is enabled by `CAP_CHOWN`.

A key insight is the observation that programs, not people, exercise privilege. That is, everything done in a computer is via agents—programs—and only if these programs know what to do with privilege can they be trusted to wield it. The UID=0 model of privilege makes privilege a feature of the super-user context, and means that it is arbitrary which programs can do privileged things. Capabilities, however, limit which programs can wield any privilege to only those programs marked with filesystem-capabilities. This feature is especially important in the aftermath of a hostile user exploiting a flaw in a setuid-*root* program to gain super-user context in the system.

This paper describes how to use the Linux implementation of these capabilities. As we will show, while support of legacy software requires that we sometimes maintain a privileged root user, the full implementation

---

[1] Details about effective, saved, and filesystem UIDs, groups, and group membership have been omitted from this discussion. That being said, through complexity, they have greatly added to the usability of the system.

[2] In practice, a shared need to edit a protected file like this can be achieved with ACLs—requiring a `shadow`-UID or group for example.

of Linux capabilities enables one to box-in certain subsystems in such a way that the UID=0 context becomes that of an unprivileged user. As legacy software is updated to be capability-aware, a fully root-less system becomes a very real possibility [12].

2.1 Capability rules

Processes and files each carry three capability sets. The process *effective* set contains those capabilities which will be applied to any privilege checks. The *permitted* set contains those capabilities which the task may move, via the `capset()` system call, into its *effective* set. The *effective* set is never a superset of the *permitted* set. The *inheritable* set is used in the calculation of capability sets at file execution time.

Capabilities are first established, at program execution time, according to the following formulas:

$$pI' = pI \quad (1)$$

$$pP' = (X \& fP) \mid (pI \& fI) \quad (2)$$

$$pE' = fE \mathbin{?} pP' : \emptyset. \quad (3)$$

Here pI, pE, and pP are the process' inheritable, effective, and permitted capability sets (respectively) before `exec()`. Post-`exec()`, the process capabilities sets become pI', pE', and pP'. The capability sets for the file being executed are fI, fE, and fP. Equation 1 shows the task retains its pre-`exec()` inheritable set. Equation 2 shows the file inheritable and process inheritable sets are and'ed together to form a context-dependent component of the new process permitted set. The file inheritable set, fI, is sometimes referred to as the file's *optional* set because the program will only acquire capabilities from it if the invoking user context includes them in pI. By optional, we mean the program can gracefully adjust to the corresponding privileges being available or not. The file permitted set, fP, is also called the *forced* set because capabilities in that set will be in the process' new permitted set whether it previously had them in any capability sets or not (subject to masking with X). In Equation 3, the file effective capability set is interpreted as a boolean. If fE (also called the *legacy* bit) is set, then the process' new effective set is equal to the new permitted set. If unset, then pE' is empty when the `exec()`d program starts executing.

The remaining object in these rules, X, has, until recently, been an unwieldy *system-wide* capability bounding set. However, it has now become the per-process capability bounding set. X is inherited without modification at `fork()` from the parent task. A process can remove capabilities from its own X so long as its effective set has `CAP_SETPCAP`. A task can never add capabilities to its X. However, note that a task can gain capabilities in pP' which are not in X, so long as they are both in pI and fI. The bounding set will be further discussed in Section 3.3.

When a new process is created, via `fork()`, its capability sets are the same as its parent's. A system call, `capset()`, can be used by a process to modify its three capability sets: pI, pP and pE. As can be seen in Equation 1, the inheritable set pI remains unchanged across file execution. Indeed it is only changed when the running process uses the system call to modify its contents. Unless pE contains `CAP_SETPCAP`, Linux will only allow a process to *add* a capabilities to pI that are present in pP. No special privilege is required to *remove* capabilities from pI. The only change to the permitted set, pP, that a process can make is to drop raised capabilities. The effective set is calculated at file execution, and immediately after `exec()` will be either equal to the permitted set or will be empty. Via `capset()` the process can modify its effective set, pE, but Linux requires that it is never a superset of the contents of the process' permitted set, pP.

Most software and distributions available currently depend on the notion of a fully privileged `root` user. Linux still supports this behavior in what we call *legacy-fixup* mode, which is actually the default. Legacy-fixup mode acts outwardly in a manner consistent with there being a `root` user, but implements super-user privilege with capabilities, and tracks UID-changes to *fixup* the prevailing capability sets. This behavior allows a root user to execute any file with privilege, and an ordinary user to execute a setuid-`root` file with privilege. When active, legacy-fixup mode force-fills the file capability sets for every setuid-`root` file and every file executed by the `root` user. By faking a full fP and full fI we turn a setuid-`root` file or a file executed by the `root` user into a file carrying privilege. This may appear distasteful, but the desire to support legacy software while only implementing one privilege model within the kernel requires it. As we will show in Section 4 legacy-fixup mode can be turned off when user-space needs no privilege or supports pure privilege through capabilities.

In the absence of VFS support for capabilities, a number of extensions to the basic capability model [9] were introduced into the kernel: an unwieldy (global, asynchronous,[3] and system crippling) bounding set [13]; the unwieldy (asynchronous and questionable) remote bestowal of capabilities by one process on another;[4] the unwieldy (global, asynchronous, and system crippling) secure-bits;[5] and the more moderately scoped `prctl(PR_SET_KEEPCAPS)` extension.

All but the last of these have recently been made viable through limiting their scope to the current process, becoming synchronous features in the Linux capability model. The `prctl(PR_SET_KEEPCAPS)` extension of legacy-fixup mode, which can be used as a VFS-free method for giving capabilities to otherwise unprivileged processes, remains so. When switching from the privileged `root` user to a non-`root` user, the task's permitted and effective capability sets are cleared.[6] But, using `prctl(PR_SET_KEEPCAPS)`, a task can request keeping its capabilities across the next `setuid()` system call. This makes it possible for a capability-aware program started with `root` privilege to reach a state where it runs locked in a non-`root` user context with partial privilege. As we discuss in Section 4, while legacy-fixup remains the default operating mode of the kernel, each of these *legacy* features can be disabled on a per-process basis to create process-trees in which legacy-fixup is neither available nor, indeed, needed.

3 Worked Examples

In this section we provide some explicit examples for how to use capabilities. The examples show how traditional setuid-`root` solutions can be emulated, and also what is newly possible with capabilities.

3.1 Minimum privilege

In this example we consider an application, `ping`, that one might not even realize requires privilege to work. If you examine the regular file attributes of a non-capability attributed `ping` binary, you will see something like this:

```
$ ls -l /bin/ping
-rwsr-xr-x  1 root root 36568 May 2 2007 /bin/ping
$ /bin/ping -q -c1 localhost
PING localhost.localdomain (127.0.0.1) 56(84)
bytes of data.
--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.027/0.027/0.027/0.000 ms,
pipe 2
$
```

The `s` bit of the file's mode is the familiar setuid-executable bit. If we copy the file as an unprivileged user (`luser`) it loses its privilege and ceases to work:

```
$ cp /bin/ping .
$ ls -l ping
-rwxr-xr-x  1 luser luser 36568 Mar 26 17:54 ping
$ ./ping localhost
ping: icmp open socket: Operation not permitted
$
```

Running this same program as `root` will make it work again:

```
# ./ping -q -c1 localhost
PING localhost.localdomain (127.0.0.1) 56(84)
bytes of data.
--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.027/0.027/0.027/0.000 ms,
pipe 2
#
```

In short, `ping` requires privilege to write the specially crafted network packets that are used to probe the network.

Within the Linux kernel there is a check to see whether this application is `capable(CAP_NET_RAW)`, which means `cap_effective` (*pE*) for the current process includes `CAP_NET_RAW`. By default, `root` gets all effective capabilities, so it defaults to having more-than-enough privilege to successfully use `ping`. Similarly, when setuid-`root`, the `/bin/ping` version is also overly privileged. If some attacker were to discover

---

[3] Asynchronocity with respect to security context means that a task's security context can be changed by another task without the victim's awareness.

[4] The ability for one process to asynchronously change, without notification, the capabilities of another process, via the *hijacked* `CAP_SETPCAP` capability, was so dangerous to system integrity that it has been disabled by default since its inception in the kernel. The addition of VFS support disables this feature and restores `CAP_SETPCAP` to its intended use as documented in this paper (see Section 3.1).

[5] Securebits have been implemented in the kernel for many years, but have also been cut off from being available—without any API/ABI for manipulating them for almost as long.

[6] The actual semantics of legacy-fixup are more complicated.

a new buffer-overflow [14] or more subtle bug in the `ping` application, then they might be able to exploit it to invoke a shell with `root` privilege.

Filesystem capability support adds the ability to bestow *just-enough* privilege to the `ping` application. To emulate just enough of its legacy privilege, one can use the utilities from `libcap` [10] to do as follows:

```
# /sbin/setcap cap_net_raw=ep ./ping
# /sbin/getcap ./ping
./ping = cap_net_raw+ep
```

What this does is add a permitted capability for `CAP_NET_RAW` and also sets the *legacy* effective bit, fE, to automatically raise this *effective* bit in the `ping` process (pE) at the time it is invoked:

```
$ ./ping -q -c1 localhost
PING localhost.localdomain (127.0.0.1) 56(84)
bytes of data.
--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet
loss, time 0ms
rtt min/avg/max/mdev = 0.093/0.093/0.093/0.000
ms, pipe 2
$
```

Unlike the setuid-`root` version, the binary `ping` is not bestowed with any privilege to modify a file that is not owned by the calling user, or to insert a kernel module, etc. That is, there is no direct way for some malicious user to subvert this *privileged* version of `ping` to do anything privileged other than craft a malicious network packet.[7]

So far, we have explained how to replace the setuid-`root` privilege of `ping` with file capabilities. This is for an unmodified version of `ping`. It is also possible to lock `ping` down further by modifying the `ping` source code to use capabilities explicitly. The key change from the administrator's perspective is to set `ping`'s capabilities as follows:

```
# /sbin/setcap cap_net_raw=p ./ping
```

That is, no *legacy* effective bit, and no enabled privilege (just the potential for it) at `exec()` time. Within the `ping` application one can, using the API provided by `libcap` [10], prepare to manipulate the application's privilege by crafting three capability sets as follows:

```
       /* the one cap ping needs */
const cap_value_t cap_vector[1] =
      { CAP_NET_RAW };
cap_t privilege_dropped = cap_init();
cap_t privilege_off = cap_dup(privilege_dropped);
cap_set_flag(privilege_off, CAP_PERMITTED, 1,
   cap_vector, CAP_SET);
cap_t privilege_on = cap_dup(privilege_off);
cap_set_flag(privilege_on, CAP_EFFECTIVE, 1,
   cap_vector, CAP_SET);
```

Then, as needed, the capability sets can be used with the following three commands:

```
/* activate: cap_net_raw=ep */
if (cap_set_proc(privilege_on) != 0)
    abort("unable to enable privilege");
/* ...do privileged operation... */
/* suspend: cap_net_raw=p */
if (cap_set_proc(privilege_off) != 0)
    abort("unable to suspend privilege");
/* ...when app has no further need of privilege */
if (cap_set_proc(privilege_dropped) != 0)
    abort("failed to irrevocably drop privilege");
```

Also, remember to clean up allocated memory, using `cap_free(privilege_on)` etc., once the capability sets are no longer needed by the application. These code snippets can be adapted for other applications, as appropriate.

In these code snippets, the inheritable capability set is forced to become empty. This is appropriate and suffices for applications that do not expect to execute any files requiring privilege, or which expect any privilege in subsequently executed programs to come from the file's forced set (fP). For an application like a user shell, the above snippets might be changed so as to preserve pI. This can be achieved by replacing the use of `cap_init()`, above, with the following sequence:

```
cap_t privilege_dropped = cap_get_proc();
cap_clear_flag(privilege_dropped, CAP_EFFECTIVE);
cap_clear_flag(privilege_dropped, CAP_PERMITTED);
```

A login process, in turn, would likely be authorized with `CAP_SETPCAP`, allowing it to actually fill pI further with specific capabilities assigned to the user being logged-in. Section 3.2 will begin to show how to use inherited privilege.

3.2 Inherited privilege

There are some programs that don't have privilege, per se, but wield it in certain circumstances: for example,

---

[7] Of course, it may prove possible to leverage a rogue network packet to cause system damage, but only indirectly—by subverting some other privileged program.

when they are invoked by `root`. One such application is `/bin/rm`. When invoked by `root`, `/bin/rm` can delete a file owned by *anyone*. Clearly, forcing privilege with the file permitted bits, as we did in the previous section, would give any invoker of `/bin/rm` such abilities and not represent an increase in security at all! To emulate `root`-is-special semantics for certain users, we employ the *inheritable* capability set (*pI*).

The basic setup for leveraging inheritable capabilities is to add file capabilities to `/bin/rm` as follows (in this case, we'll add the capability to the official `rm` binary):

```
# /sbin/setcap cap_dac_override=ei /bin/rm
```

Reviewing the capability formula, Equation 1, one can see that a process inherits its *inheritable* capabilities, *pI*, directly from its parent. In order to use inheritable capabilities, therefore, a process has to first acquire them. The `libcap` package provides a utility for reading the capabilities of a process:

```
$ /sbin/getpcaps 1
Capabilities for `1': =ep cap_setpcap-e
$
```

This says that `init`, the top of the process tree, and ancestor to all processes in a system, does not have any *inheritable* capabilities. That is, by default, no process will passively obtain any inheritable capabilities. However, `init` and its many privileged descendants, such as `login` and `su`, do have access to capabilities through their *permitted* sets, *pP*. To add a capability to its inheritable set, a process must either have that capability present in its *permitted* set, or be `capable(CAP_SETPCAP)`—have the single capability, `CAP_SETPCAP` in its *effective* set, *pE*. Leveraging this feature, the `libcap` package [10] contains two convenient methods to introduce inheritable capabilities to a process-tree: a simple wrapper program, `capsh`, and a PAM [15] module, `pam_cap.so`.

The `capsh` command is intended to provide a convenient command-line wrapper for testing and exploring capability use. It is able to alter and display capabilities of the current process and can be used to explore the nuances of the present example. We shall use `capsh` in Section 3.3. Here we will describe how to make use of the `pam_cap.so` PAM module.

The PAM module `pam_cap.so`, as directed by a local configuration file, sets inheritable capabilities based on the user being authenticated. In our example, we give a student administrator (`studadmin`) the ability to remove files owned by others. We set up a test file and a configuration file (as `root`) with the following commands:

```
# cat > /etc/security/su-caps.conf <<EOT
cap_dac_override    studadmin
none                *
EOT
# touch /etc/empty.file
```

```
# ls -l /etc/{empty.file,security/su-caps.conf}
-rw-r--r--  1 root root  0 Mar 30 14:00 /etc/empty.file
-rw-r--r--  1 root root 52 Mar 30 13:59 /etc/security/su-caps.conf
```

We then put the following line at the very beginning of the `/etc/pam.d/su` file:

```
auth optional  pam_cap.so \
  config=/etc/security/su-caps.conf
```

Now anyone able to authenticate via `su studadmin` will become the regular user `studadmin` with the enhancement that they have an inheritable capability, `CAP_DAC_OVERRIDE`:

```
$ whoami
luser
$ su studadmin
Password:
$ whoami
studadmin
$ /sbin/getpcaps $$
Capabilities for `11180': = cap_dac_override+i
$
```

Having obtained this inheritable capability, `studadmin` can try it out by deleting a `root`-owned file:

```
$ rm /etc/empty.file
rm: remove write-protected regular
file `/etc/empty.file'? y
$ ls -l /etc/empty.file
ls: /etc/empty.file: No such file or directory
```

In passing, we note that when the `rm` command was prompting for the `y` response, it was possible to find the PID for this process and, from a separate terminal:

```
$ /sbin/getpcaps 15310
Capabilities for `15310': = cap_dac_override+eip
$
```

That is, observe that the formula Equation 2 did its work to raise the permitted, *pP*, capability for `rm`, and the legacy fE bit caused it to become effective for the process at `exec()` time.

It is instructive to try to remove something else using another program. For example, using `unlink`:

```
$ unlink /etc/security/su-caps.conf
unlink: cannot unlink
'/etc/security/su-caps.conf': Permission denied
$
```

Because this `unlink` application has no filesystem capabilities, $fI = fP = fE = 0$, despite the prevailing inheritable capability in *pI*, `unlink` cannot wield any privilege. A key feature of the capability support is that only applications bearing filesystem capabilities can wield any system privilege.

In this example, we have demonstrated how legacy applications can be used to exercise privilege through inheritable capabilities. As was the case in the previous example, legacy applications can be modified at the source code level, to manipulate capabilities natively via the API provided by `libcap`. Such a modified application would not have its legacy capability raised ($fE = 0$). The code samples from the previous section are equally applicable to situations in which an application obtains its capabilities from its inheritable set, we do not repeat them here.

3.3 Bounding privilege

The capability bounding set is a per-process mask limiting the capabilities that a process can receive through the file permitted set. In Equation 2, the bounding set is X. The bounding set also limits the capabilities which a process can add to its *pI*, though it does not automatically cause the limited capabilities to be removed from a task which already has them in *pI*. When originally introduced [13], the capability bounding set was a system-wide setting applying to all processes. An example intended usage would have been to prevent any further kernel modules from being loaded by removing `CAP_SYS_MODULE` from the bounding set.

Recently, the bounding set became a per-process attribute. At `fork()`, a child receives a copy of its parent's bounding set. A process can remove capabilities from its bounding set so long as it has the `CAP_SETPCAP` capability [16]. Neither a process itself, nor any of its `fork()d` children, can ever add capabilities back into its bounding set. The specific use case motivating making the bounding set per-process was to permanently remove privilege from containers[17] or jails[18]. For instance, it might be desirable to create a container unable to access certain devices. With per-process capability bounding sets, this becomes possible by providing it with a `/dev` that does not contain these devices and removing `CAP_MKNOD` from its capabilities.[8]

The reader will note, in Equation 2, that X masks only fP. In other words, a process' permitted set can receive capabilities which are not in its bounding set, so long as the capabilities are present in both fI and *pI*. Ordinarily this means that a process creating a "secure container" by removing some capabilities should take care to remove the unwanted capabilities from both its bounding and inheritable sets. Thereafter they cannot be added back to *pI*. However, there may be cases where keeping the bits in the inheritable and not the bounding set is in fact desirable. Perhaps it is known and trusted that the capability will only be in fI for trusted programs, so any process in the container executing those programs can be trusted with the privilege. Or, the initial container task may take care to spawn only one task with the capability in its *pI*, then drop the capability from its own *pI* before continuing. In this way the initial task in a container without `CAP_MKNOD`, rather than mounting a static `/dev`, could keep `CAP_MKNOD` in *pI* while running a trusted copy of `udev`, from outside the container, which has `CAP_MKNOD` in its fI. The `udev` process becomes the only task capable of creating devices, allowing it to fill the container's `/dev`.

Here is an example of dropping `CAP_NET_RAW` from the bounding set, using `capsh` [10]. So doing, we can cause `ping` to fail to work as follows:

```
# id -nu
root
# /sbin/getcap ./ping
./ping = cap_net_raw+ep
# /sbin/capsh --drop=cap_net_raw \
  --uid=$(id -u luser) --
$ id -nu
luser
$ ./ping -q -c1 localhost
ping: icmp open socket: Operation not permitted
$ /bin/ping -q -c1 localhost
ping: icmp open socket: Operation not permitted
```

---

[8] This requires a (hopefully) upcoming patch causing mounts by a process which is not `capable(CAP_MKNOD)` to be `MNT_NODEV`.

The `--drop=cap_net_raw` argument to `/sbin/capsh` causes the wrapper program to drop `CAP_NET_RAW` from the bounding set of the subsequently invoked `bash` shell. In this process tree, we are unable to gain enough privilege to successfully run `ping`. That is, both our capability-attributed version, and the setuid-`root` version attempt to force the needed privilege, but the prevailing bounding set, X, suppresses it at execution time.

In an environment in which the bounding set suppresses one or more capabilities, it is still possible for a process to run with these privileges. This is achieved via use of the inheritable set:

```
# id -nu
root
# /sbin/setcap cap_net_raw=eip ./ping
# /sbin/capsh --{inh,drop}=cap_net_raw \
  --uid=$(id -u luser) --
$ ./ping -q -c1 localhost
PING localhost.localdomain (127.0.0.1) 56(84)
bytes of data.
--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.037/0.037/0.037/0.000 ms,
pipe 2
```

That is, as per Equation 2, the bounding set, X, does not interfere with the $pI \& fI$ component to pP'.

There are some subtleties associated with bounding set manipulation that are worth pointing out here.

The first is that the bounding set does limit what capabilities can be *added* to a process' inheritable set, pI. For example, as `root`:

```
# /sbin/capsh --drop=cap_net_raw --inh=cap_net_raw
Unable to set inheritable capabilities:
Operation not permitted
#
```

This fails because, by the time we attempt to add an inheritable capability in the working process, we have already removed it from the bounding set. The kernel is just enforcing the rule that once pI and X are *both* without a particular capability, it is irrevocably suppressed.

The second subtlety is a warning, and relates to a bug first highlighted in association with the *sendmail* program [16]. Namely, for *legacy* programs that require forced capabilities to work correctly, you can cause them to fail in an unsafe way by selectively denying them privilege. When a legacy program makes the (common) assumption that an operation must work because the program is known to be operating with privilege (a previous privileged operation has succeeded), with capabilities, it can be fooled into thinking it is operating in one privilege level when it actually isn't. Since privilege is now represented by independent capabilities, one can leverage the bounding set to deny a single capability that is only needed later at a more vulnerable time in the program's execution.

The *sendmail* issue was in a context where the dropping of an inheritable capability by an unprivileged parent of the setuid-`root` sendmail caused sendmail to launch a program as `root` when it thought it was running in the context of the `luser`. The significance of the bug was that an unprivileged `luser` could exploit it.

The kernel was fixed to make this particular situation not occur. However, the bounding set actually recreates a similar situation, and while sendmail has since been fixed to protect it from this problem, many other legacy setuid-`root` applications are expected to suffer from this same issue. Non-legacy applications are not susceptible to this subtlety because they can leverage the `libcap` API to look-before-they-leap and check if they have the needed privilege explicitly at runtime.

The significant difference between the old problematic situation and this present case, is that to exploit this issue you need to be able to alter the bounding set and that, itself, requires privilege. That being said, this subtlety remains. Be careful, when using the bounding set, to avoid leveraging it suppress privilege in general when it is more appropriate to supply *optional* capabilities as needed via the inheritable set. Caveat emptor!

3.4 No privilege at all

In general, unprivileged users need to run privileged applications. However, sometimes it may be desirable to confine a process, and any of its children, ensuring that it can never obtain privilege. In a traditional UNIX system this would not be possible, as executing a setuid-`root` program would always raise its privilege.

To completely remove privilege from a process in a capability-enabled Linux system, we must make sure that both sides of Equation 2 are, and always will be, empty. We can suppress fP by emptying the bounding set, X. Since a capability can never be added back into X, this is irrevocable. Next, we can suppress the second half of the equation by emptying *pI* using `capset`. Thereafter the process cannot add any bits not in X (which is empty), back into *pI*. The legacy compatibility mode refills fI whenever a setuid root binary is executed, but we can see in Equation 2 that capabilities must be in both fI and *pI* to appear in pP'. Now, regardless of what the process may execute, neither the process nor any of its children will ever be able to regain privilege.

4 Future changes

At the conclusion of Section 2.1 we observed that the capability rules are perverted for files run by the super-user. When the super-user executes a file, or when any user executes a setuid-`root` file, the file's capability sets are filled. Since historically Linux had no support for file capabilities, and since without file capabilities a process can never wield privilege in a pure capability system, this *hack* was unfortunate but necessary. Now that the kernel supports file capabilities, it is only userspace which must catch up. As applications become capability-aware, it will be desirable to remove the legacy `root`-as-super-user support for those applications. While infrastructure to support disabling it system-wide has been present for as long as the `root`-as-super-user hack has existed, support to do this for application sets has only recently been accepted into the experimental `-mm` tree [19]. It is expected to be adopted in the main Linux tree [20], and may have done so by the time of publication.

With the per-process *securebits*, the root user exception can be "turned off" for capability-aware applications by setting the `SECURE_NOROOT` and `SECURE_NO_SETUID_FIXUP` flags using `prctl()`. These are per-process flags, so that a system can simultaneously support legacy software and capability-aware software. In order to lock capability-aware software into the more secure state in a such a way that an attacker cannot revert it, both bits can be locked by also setting `SECURE_NOROOT_LOCKED` and `SECURE_NO_SETUID_FIXUP_LOCKED`.

To nail the residue of problematic partial privilege for legacy applications, discussed in Section 3.3, we are considering adding a requirement that any legacy application which is made privileged with $fE \neq 0$ must execute with $pP' \geq fP$. That is, if the bounding set, X, suppresses a *forced* capability ($fP < fP\&X$), and the inheritable sets ($pI\&fI$) do not make up for its suppression (see Equation 2), `exec()` will fail with `errno = EPRIV`. This change will enforce what is presently only a convention that legacy applications should run with all of their *forced* (fP) capabilities raised, or are not safe to run at all.

5 Conclusion

The intent of this paper has been to demonstrate that the Linux *capability* implementation, with VFS support, is a viable privilege mechanism for the Linux kernel. With examples, we have shown how these capabilities can and should be used. What remains is for user-space applications to start using them.

That being said, privilege is not the only use of the `root` identity. There are many files, such as are to be found in `/proc/` and `/etc/`, that are owned by `root`. Even without super-user privilege, a process running in the context of an impotent `root` user, can still do a large amount of damage to a system by altering these files. Here, DAC and MAC based security will continue to be important in securing your Linux system.

Acknowledgments

It is our pleasure to thank Chris Friedhoff and KaiGai Kohei for collaboration; Chris Wright, James Morris, and Stephen Smalley for advice and careful scrutiny; Olaf Dietsche and Nicholas Simmonds for posting alternative implementations of file capabilities; and Andrew Morton for patiently sponsoring the recent capability enhancements to the kernel. AGM would additionally like to thank Alexander Kjeldaas, Aleph1, Roland Buresund, Andrew Main and Ted Ts'o for early `libcap` and kernel work; the anonymous chap that let AGM read a copy of a POSIX draft sometime around 1998; and Casey Schaufler for persuading the POSIX committee to release an electronic copy of the last POSIX.1e draft [9]. Finally, we'd like to thank Heather A Crognale for her insightful comments on an early draft of this paper.

References

[1] Andreas Grünbacher. POSIX Access Control Lists on Linux. USENIX Annual Technical Conference, San Antonio, Texas, June 2003.

[2] Chris Wright et al. Linux Security Modules: General Security Support for the Linux Kernel. 11th USENIX Security Symposium, 2002.

[3] Casey Schaufler. The Simplified Mandatory Access Control Kernel. linux.conf.au, 2008.

[4] W.E. Boebert and R.Y. Kain. A practical alternative to hierarchical integrity policies. In Proceedings of the Eighth National Computer Security Conference, 1985.

[5] Peter Loscocco, Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 2001.

[6] The Trusted Computer System Evaluation Criteria (*the Orange Book*). `http://www.fas.org/irp/nsa/rainbow/std001.htm`.

[7] Samuel Samalin, Secure UNIX, McGraw-Hill, 1996.

[8] Whole system compromises are regularly documented on websites such as *Bugtraq*: `http://www.securityfocus.com/archive/1`.

[9] The last POSIX.1e draft describing capabilities: `http://wt.xpilot.org/publications/posix.1e/download.html`.

[10] The capability user-space tools and library: `http://www.kernel.org/pub/linux/libs/security/linux-privs/libcap2/`.

[11] See for example: `http://people.redhat.com/sgrubb/audit/`.

[12] Advocates for transforming systems to be capability based such as: `http://www.friedhoff.org/fscaps.html`.

[13] Introduction of the *global* bounding set to Linux, `http://lwn.net/1999/1202/kernel.php3`.

[14] An example of a buffer-overflow in the `ping` binary (not exploitable in this case): `http://www.securityfocus.com/bid/1813`.

[15] Linux-PAM, `http://www.kernel.org/pub/linux/libs/pam`; Kenneth Geisshirt, Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers. Packt Publishing. 2006.

[16] Discussion of the unfortunately named *sendmail-capabilities-bug*: `http://userweb.kernel.org/~morgan/sendmail-capabilities-war-story.html`.

[17] Linux Containers, `http://lxc.sourceforge.net/`.

[18] Poul-Henning Kamp, Robert N.M. Watson: Jails: Confining the omnipotent root. Proceedings of second international SANE conference, May 2000.

[19] Andrew Morton's kernel patch series: `http://www.kernel.org/patchtypes/mm.html`

[20] Linus' kernel tree: `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary`

Proceedings of the Linux Symposium

Volume One

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*