

Overview of LIDS, Part Two

Brian Hatch 2001-10-31

Overview of LIDS, Part Two

by *Brian Hatch*

last updated October 31, 2001

This is the second part of a four-part series devoted to an overview of LIDS, a Linux kernel patch that will allow users to take away the all-powerful nature of root in order to give programs exactly the access they need and no more. The [first article in this series](#) offered an overview of LIDS. This installment will look at file restrictions, LIDS File ACLs, and LIDS enhancements of Linux capabilities.

File Restrictions

Linux supports two forms of file permission protection by default. The first are your standard Unix user/group/other permissions, set with the `chown` and `chmod` command. Take the following file as an example.

```
user$ ls -l rotatweblogs
-rwsr-xr--  1 root      web          14854 Sep 17 02:21 rotateweblogs
```

This file is owned by root:web, can be read by any user, run by any user in the group web, and will have superuser privileges when executed (in this case, presumably to send a USR1 signal to apache after rotating the log files.)

The second protection for files is based on the filesystem itself. For example the ext2 filesystem, the default for most Linux distributions, can set extended attributes with the `chattr` command, for example:

```
root# lsattr /var/log/messages
----- /var/log/messages
root# chattr +a /var/log/messages
root# lsattr /var/log/messages
-----a-- /var/log/messages
```

Here we've set the 'append only' attribute to `/var/log/messages`. Other ext2 attributes include:

```
+i (immutable, cannot change or delete file)
+s (file will have it's contents erased from disk when the file is
deleted)
+d (file cannot be archived with the dump command).
```

LIDS File ACLs

LIDS uses its own brand of ACLs directly in the kernel; they are integrated into the VFS (Virtual File System) layer, and thus do not depend on the filesystem type itself. This means that they could apply equally well to ext2, ext3, reiserfs, or even remotely-mounted partitions. (The ACLs enabled for remotely-mounted partitions would only apply to machines running LIDS, not to any other machines that mount the same filesystems.)

LIDS File ACLs are enforced as soon as the machine boots, and cannot be turned off unless you turn off LIDS entirely (`lidsadm -S -- -LIDS_GLOBAL.`) This can cause some headaches on machines with newly-installed LIDS until all the file ACLs are perfected.

There are four kinds of ACLs that LIDS can enforce on files. They are as follows:

- DENY - the existence of the object is flatly denied. Any program attempting to access it won't find it.
- READ - the object is available read only and cannot be modified in any way.
- APPEND - the object is available for reading and for writing in append mode only. This is very useful for log files and other files that can grow but should not be otherwise modified.
- WRITE - the object is unprotected, as if LIDS were not in use at all.

LIDS ACLs can be placed on individual files, in which case the ACL applies to that file only, or on directories, in which case the ACL applies to all files in that directory recursively. The ACL can be defined with the `lidsadm` command, such as

```
lfs# lidsadm -A -o /etc -j READ
```

(Remember, we must be in a LIDS Free Session to use the `lidsadm` command.) This adds (-A) an ACL that sets /etc and all files therein to be available as read only. You cannot change or add files in /etc at all:

```
root# echo "testing" > /etc/testing
cannot create /etc/blah: Read-only file system
```

However, you can set additional ACLs if you wish as well to tailor the behavior of files in /etc, such as

```
lfs# lidsadm -A -o /etc/motd -j WRITE

lfs# lidsadm -A -o /etc/shadow -j DENY
```

The more specific ACL takes precedence. Thus /etc/motd is available for writing even though /etc is

read only. The file `/etc/shadow` is unavailable entirely, as we can see here:

```
root# echo "Testing" >> /etc/motd; tail -1 /etc/motd
Testing

root# ls -l /etc/shadow
ls: /etc/shadow: No such file or directory
```

You may be wondering what purpose a hidden `/etc/shadow` serves. Sure, it means that even if a cracker gets root access they cannot read the password file and attempt to crack passwords. However, it also means that legitimate system programs such as `login`, `sshd`, and other authentication systems cannot verify passwords at all either. This is where LIDS ACL Subjects come into play.

In addition to specifying an object (`/etc/shadow` above,) you can specify a subject to the `lidsadm` command. If we want to make `/etc/shadow` unreadable except for `sshd`, we'd use the following two commands:

```
lfs# lidsadm -A -o /etc/shadow -j DENY
lfs# lidsadm -A -s /usr/sbin/sshd -o /etc/shadow -j READ
```

This gives `sshd` permission to read the shadow file, and denies access to all other programs. `Sshd` still only has `READ` permission, not full `WRITE` permission, so we have given it just enough access to function as needed. You would use a similar command to give other authentication programs, such as `login`, `sulogin`, `su`, `sudo`, and `vlock`, `READ` access to `/etc/shadow`.

When faced with multiple rules that could apply to a specific access, LIDS picks the most specific. Thus in our shadow example we have three rules which are in use, from most specific to most general:

```
/usr/sbin/sshd has READ access to /etc/shadow.
All files have no (DENY) access to /etc/shadow.
All files have READ access to files in /etc.
```

Protect All Subjects

Say you have an object that is protected, to which you wish to give a program special access; for example, protecting all `/var/log` files, but giving `sshd` the ability to modify `/var/log/wtmp`:

```
lfs# lidsadm -A -o /var/log -j APPEND
lfs# lidsadm -A -s /usr/sbin/sshd -o /var/log/wtmp -j WRITE
```

LIDS allows you to create the second rule only if `/usr/sbin/sshd` is also protected. If you're protecting everything in `/usr` or `/usr/sbin`, then you're fine already. However, if you are not that broad in your

READ permissions, you would need to first protect `sshd` before you can create equivalent rules, as seen here:

```
lfs# lidsadm -A -o /var/log -j APPEND
lfs# lidsadm -A -o /usr/sbin/sshd -j READ
lfs# lidsadm -A -s /usr/sbin/sshd -o /var/log/wtmp -j WRITE
```

Since `sshd` is being given access to an otherwise protected file, LIDS requires that `sshd` also be protected.

Protect Your Filesystem Devices

Your filesystems are mounted from block devices in the `/dev` directory, such as `/dev/hda1` or `/dev/sda3`. While the standard method to access them is through the mount point itself, it is still possible to read and write to the block device directly.

If someone can access the raw device files, they can circumvent the File ACLs enforced by LIDS. Thus it is important to prevent this. True, you're unlikely to find a script-kiddie smart enough to muck with your raw disks, but it's best not to underestimate the enemy.

You should remove the capability `CAP_SYS_RAWIO` from the bounding set by making sure that it starts with a '-' in `lids.cap`:

```
-17:CAP_SYS_RAWIO
```

This automatically prevents any raw io access to devices. The only program you're likely to need that requires raw access to devices is your X server, in order to work with your video card and monitor, so if you need X make sure to grant it this capability:

```
lfs# lidsadm -A -s /path/to/X_server -o CAP_SYS_RAWIO -j GRANT
```

LIDS ACLs Are Inode-Based

If you look at the `lids.conf` file, you will notice the inode of the file being referenced, as well as the major and minor device number of the filesystem on which it resides. LIDS maintains the ACLs internally using the filesystem and inode number, rather than by the pathname itself.

This has both positive and negative effects. If you have a file with more than one hardlink, each file is protected with the exact same ACLs because they share an inode. However, it also means that changing a file's inode breaks the ACL association.

The most common time you will have inode changes are when programs are deleted and recreated (for example `/etc/passwd` may get copied, modified, and the new version installed after it is verified to be valid) or when software installations or upgrades are performed. In order to make sure that your ACLs are pointing to the correct files, you must upgrade the LIDS inode table when changes occur:

```
lfs# lidsadm -U
```

This updates the `lids.conf` file to match the current state of the filesystems and files being protected. In order to update the LIDS kernel data structures, make sure to reload the configuration files with

```
lfs# lidsadm -S -- +RELOAD_CONF
```

Linux Capabilities

In traditional Unix models, the root user is all-powerful. An attempt was made to create a standard method of categorizing the various access checks, such as file permission override, network configuration, raw device access, and so forth, into a discrete set of groups called capabilities. A vendor-independent committee worked to standardize capabilities across Unix platforms in a draft designated POSIX 1003.1e. Unfortunately, the committee gave up and draft never became an actual standard.

Linux includes a variant of this capability model. Instead of checking for `UID==0` in kernel access control checks, the kernel uses the `'capable()'` call. This call will check the capabilities granted to a process to determine if it should have privileged access. The capabilities on a standard Linux kernel are listed here with a very brief explanation of the purpose. (For more detailed description, see `/usr/include/linux/capability.h` on your system):

<code>CAP_CHOWN</code>	Allow unrestricted use of <code>chown</code> to change file ownership
<code>CAP_DAC_OVERRIDE</code>	Allow unlimited file access (No DAC restrictions.)
<code>CAP_DAC_READ_SEARCH</code>	Allow all read/search related actions regardless of file permissions.
<code>CAP_FOWNER</code>	Allow file access even when <code>owner-id != userid</code>
<code>CAP_FSETID</code>	Allow the setting of <code>setuid/setgid</code> flags on any file.
<code>CAP_KILL</code>	Allow signals to be sent to processes you don't own.
<code>CAP_SETGID</code>	Allow unrestricted <code>setgid(2)</code> and <code>setgroups(2)</code> .
<code>CAP_SETUID</code>	Allow unrestricted <code>setuid(2)</code> and friends.
<code>CAP_SETPCAP</code>	Allow you to transfer any capability you possess to another PID.
<code>CAP_LINUX_IMMUTABLE</code>	Allow modification of immutable and append file attributes.
<code>CAP_NET_BIND_SERVICE</code>	Allow binding of TCP and UDP ports below 1024.

CAP_NET_BROADCAST	Allow outbound broadcast packets.
CAP_NET_ADMIN	Allow many options related to network interfaces, such as routing table modification, etc.
CAP_NET_RAW	Allow use of raw and packet sockets. (For hand-crafted packets, for example.)
CAP_IPC_LOCK	Allow locking of shared memory segments.
CAP_IPC_OWNER	Allow unrestricted IPC access.
CAP_SYS_MODULE	Allow the insertion and removal of LKMs.
CAP_SYS_RAWIO	Allow raw access to devices (such as /dev/[hs]da*).
CAP_SYS_CHROOT	Allow use of chroot(2).
CAP_SYS_PTRACE	Allow use of process trace of any process.
CAP_SYS_PACCT	Allow configuration of process accounting systems.
CAP_SYS_ADMIN	Allow many restricted activities such as setting hostname, using mount, creating devices, etc. (See capability.h for full list.)
CAP_SYS_BOOT	Allow use of reboot(2).
CAP_SYS_NICE	Allow priorities to be raised, and affect non-owned processes nice level.
CAP_SYS_RESOURCE	Allow access uninhibited by resource/quota/etc limits.
CAP_SYS_TIME	Allow clock manipulation.
CAP_SYS_TTY_CONFIG	Allow tty device configuration.
CAP_HIDDEN	A LIDS-specific capability, used to hide a process from /proc (ps, etc)
CAP_INIT_KILL	A LIDS-specific capability, used to limit signals to processes owned by init (daemons, usually).

In a default Linux kernel, all-powerful superuser ability is set up simply by granting all the above capabilities to any process that has uid/euid zero. All other processes do not have any capabilities set, and will be granted access based on standard Unix logic, such as file permissions and uid restrictions.

By default, all the capabilities are available to root-owned processes. However, capabilities can be removed from the kernel's capabilities bounding set itself. Doing so means that no processes can ever use that capability again until the machine is rebooted. This is a sure-fire way to remove some of the power of the root user that you can use even with standard Linux kernel. While this is one way to remove some of the power from root, it is a drastic one.

Lids Enhancements To Linux Capabilities

Lids extends the existing capabilities model in two distinct ways. As with a standard kernel, you can

remove a capability from the capability bounding set, meaning that even uid/euid root programs cannot use that capability. However, it is not removed permanently, and you can turn it back on if needed.

The second difference is that you can grant or deny specific capabilities to processes on a program-by-program basis. This allows you to finely tailor the capabilities that will be available on your system.

Lids.cap

The file `/etc/lids/lids.cap` lists the capability bounding set for the kernel. The format of the file is

```
[+-]    #:    Capability_Name
```

A snippet of this file may look like this:

```
-5:CAP_KILL
+6:CAP_SETGID
+7:CAP_SETUID
+8:CAP_SETPCAP
-9:CAP_LINUX_IMMUTABLE
-10:CAP_NET_BIND_SERVICE
```

If the first character in the line is a '+' then the capability is left in the bounding set and processes that are running as root have the capability available. If the character is a '-' then it is not available to any processes. (The number after the + or - is simply the numerical equivalent of the named capability.)

So in the above example, no processes even running as root can use `CAP_KILL`, `CAP_LINUX_IMMUTABLE`, or `CAP_NET_BIND_SERVICE`. Other capabilities are unaffected.

Capabilities Are Enforced At Sealing Time

The `lids.cap` file is not in affect when the machine is booted. This allows programs in `/etc/rcX.d` to run without any capability-related restrictions. Since files run at startup generally perform actions that require a variety of capabilities, it would be inconvenient to create exceptions to them all. Instead, the capability bounding set is enforced when the kernel is 'sealed' with the `'lidsadm -l'` command (usually as the last thing run from `/etc/rcX.d` at bootup.)

Once the kernel is sealed, all capability restrictions from `/etc/lids/lids.cap` are put in effect. You can see the current bounding set using `lidsadm` as follows:

```
lfs# lidsadm -V
VIEW
```

```
CAP_CHOWN 1
CAP_DAC_OVERRIDE 0
CAP_DAC_READ_SEARCH 0
CAP_FOWNER 1
. . . .
```

The '-V' option is only available if you compiled the lidsadm program with 'make VIEW=1'. Since it's no real security risk to allow this functionality, I prefer to compile lidsadm in this manner.

Next Time...

This concludes the second installment in this four-part series on LIDS. In the next installment, we shall continue the discussion of capabilities, looking particularly at granting capabilities and capability exceptions. We shall also look at setting ACLs and interpreting logs in LIDS.

To read **An Overview of LIDS, Part Three**, click [here](#).

Brian Hatch is an obsessive security freak and lead author of [Hacking Linux Exposed](#) and co-author of [Building Linux VPNs](#). While he frequently stays up late to write or hack code, he thinks it's much more fun to go to the park and push his daughter in the swing as he delivers horrible puns to his fiancée.

Relevant Links

[An Overview of LIDS, Part One](#)
Brian Hatch

[Privacy Statement](#)

Copyright 2006, SecurityFocus