

Overview of LIDS, Part Three

Brian Hatch 2001-11-12

Overview of LIDS, Part Three

by Brian Hatch

last updated November 12, 2001

This is the third part of a four-part article devoted to the exploration of LIDS, a Linux kernel patch that will allow users to take away the all-powerful nature of root. The first article in this series offered an overview of LIDS. The second installment looked at file restrictions, LIDS File ACLs, and LIDS enhancements of Linux capabilities. This installment will discuss granting capabilities, the LIDS-specific capabilities, ACL inheritance and time-based ACLs.

Granting Capability Exceptions

In the traditional Linux model, a capability that is removed from the bounding set is not available to any process thereafter. LIDS, however, gives us a way to grant access to capabilities on a per program basis. Thus we could remove a capability from general usage and provide it only to the functions that need it.

As an example, let's take `getty`, which is used for logging in via the console or a serial port. It requires the `CAP_SETUID` and `CAP_SETGID` capabilities to become the user that logs in, and `CAP_FOWNER`, `CAP_CHOWN`, and `CAP_DAC_OVERRIDE` to muck with the `tty` and related file permissions.

If you removed these capabilities from the system by putting a '-' in front of them in `lids.cap` then `getty` would not be able to perform these actions (and neither would any other program). We will grant a capability (the Object) to a particular program (the Subject) using the `lidsadm` command. Assuming our `getty` binary is `/sbin/getty`, we'd run the following:

```
lfs# lidsadm -A -s /sbin/getty -o CAP_SETUID -j GRANT
lfs# lidsadm -A -s /sbin/getty -o CAP_SETGID -j GRANT
lfs# lidsadm -A -s /sbin/getty -o CAP_CHOWN -j GRANT
lfs# lidsadm -A -s /sbin/getty -o CAP_FOWNER -j GRANT
lfs# lidsadm -A -s /sbin/getty -o CAP_DAC_OVERRIDE -j GRANT
```

`Getty` will be given the named capabilities even though they are removed from the bounding set. As another example, say we want to be able to synchronize our clock using `ntpd`, the following rules will grant the capabilities we need:

```
lfs# lidsadm -A -s /usr/sbin/ntpd -o CAP_NET_BIND_SERVICE -j GRANT
lfs# lidsadm -A -s /usr/sbin/ntpd -o CAP_SYS_TIME -j GRANT
```

In this case `CAP_NET_BIND_SERVICE` lets us bind a low numbered port (123) and `CAP_SYS_TIME` lets us change the system clock.

Special Capabilities

While integrating itself into the Linux kernel, LIDS took the opportunity to add and enhance some capabilities to give us an extra layer of security and configurability.

Special Handling Of CAP_BIND_NET_SERVICE

Normally when a program has CAP_BIND_NET_SERVICE capability it is able to bind any port less than 1024. LIDS has extended this capability to allow us to define which port or ports a program can bind. To use the normal behaviour, you simply grant the service as follows:

```
lfs# lidsadm -A -s /usr/sbin/httpd -o CAP_BIND_NET_SERVICE -j GRANT
```

However, if you wanted to restrict Apache to bind only port 80 (HTTP), you'd use the following:

```
lfs# lidsadm -A -s /usr/sbin/httpd \  
-o CAP_BIND_NET_SERVICE 80-80 -j GRANT
```

In order to allow it to bind port 443 (HTTPS) as well, you would use:

```
lfs# lidsadm -A -s /usr/sbin/httpd \  
-o CAP_BIND_NET_SERVICE 80-80,443-443 -j GRANT
```

You can grant a range of ports using '-', and a list of ports by joining ranges with ','. For example, if we want to allow Stunnel to listen to IMAPS (993), IRCS (994), POP3S (995) and SSMTP (465) to provide SSL services, you'd have the following:

```
lfs# lidsadm -A -s /usr/sbin/stunnel \  
-o CAP_BIND_NET_SERVICE 465-465,993-995 -j GRANT
```

This fine granular control allows you to grant programs the ability to bind only the ports they actually need, rather than using the sledgehammer approach.

CAP_INIT_KILL

The LIDS-specific capability CAP_INIT_KILL allows you to prevent daemons from being sent any signals. If you turn off this capability (by putting a '-' in front of the line in lids.cap) then no user or program will be able to signal a system daemon, where daemon is defined as a program who is a child of the init process (pid==1).

This is an excellent way to make sure that an attacker is not able to stop your log analysis tools, syslog daemon, cron, or other important services. The drawback is that you too cannot kill these programs. In fact, you can't send them any signals whatsoever, such as sending Apache a HUP to reread its config file after a

legitimate change. To send any signal to daemons you will need to be in an LFS.

Also, some daemons send signals to their parent or children, Apache being an example, so you may need to grant these processes the CAP_INIT_KILL capability as follows:

```
lfs# lidsadm -A /usr/sbin/httpd \
-o CAP_INIT_KILL -j GRANT
```

I tend to like disabling CAP_INIT_KILL in lids.conf, but it does require that extra step to use an LFS in times of need.

CAP_HIDDEN

CAP_HIDDEN is another LIDS-specific capability that is useful for hiding processes. Any program that is given the CAP_HIDDEN capability will not be visible in /proc (and thus not available to ps, top, and friends) at all.

This is not a 100% sure way of hiding processes, however. If you hide sshd, for example, folks will still be able to use 'netstat -na' and see that something is listening on port 22. A user could attempt to send a signal to each process ID from 1-65535 and determine that a hidden process existed when the signal was rejected or sent but no /proc entry exists for it. Or it could be discovered if it logs its PID to /var/log/NAME.pid.

However, I still find it useful, and like to hide some system processes with it, mainly log analysis tools, sniffers, and IDS systems.

Plus it's a nice trick to play on script kiddies.

ACL Inheritance

One of the common confusions when setting up your ACLs is why things aren't working as you think they should. For example, you may have given a script access to /etc read/write, and yet it's not functioning properly. Say you have a program /opt/bin/setmotd like the following which changes the /etc/motd file every night:

```
#!/bin/sh
/bin/cp /etc/motd.real /etc/motd
/bin/echo >>/etc/motd
/bin/uname -a >> /etc/motd
/bin/echo "It's `date +%Y/%m/%d`" >>/etc/motd
/bin/echo ' -- do you know where your $PPID is?' >> /etc/motd
```

You've set it up to run via cron, and you give it access to /etc/motd as follows:

```
lfs# lidsadm -A -s /opt/bin/setmotd -o /etc/motd -j WRITE
```

However, when it runs, all you see in `/etc/motd` are the `uname` and `date` commands, and you get the following error in your syslogs:

```
Sep 17 11:20:16 hostname kernel: LIDS: cp (3 2 inode 1037) pid 12680
  user (0/0) on pts2: Try to open /etc/motd for writing,flag=33346
```

Now you've specifically given `/opt/bin/setmotd` the ability to `WRITE` to `/etc/motd`. However the script calls `/bin/cp` to copy the header file (`/etc/motd.real`) first, and this is what is failing. The redirect (`>>`) is done by the shell script itself, and it works as expected.

The problem here is that ACLs are not inherited by their children. When `/bin/cp` is called, it is the program attempting to rewrite `/etc/motd`, and it does not have permission. In order to have your `/opt/bin/setmotd` program work, you need to set the inheritance level of the ACL. (Yes, you could rewrite the script, but there are many cases where you need to set inheritance levels of programs where you'd much rather not rewrite and recompile the software.)

LIDS ACLs can include an inheritance level to allow a subject (program) to inherit the ACL from it's parent. This is done with the `'-i'` option to `lidsadm`:

```
lfs# lidsadm -A -s /opt/bin/setmotd -o /etc/motd -j WRITE -i 1
```

In the above command we set the inheritance level to 1, meaning that `setmotd` and its children (in this case `/bin/cp`) are allowed to `WRITE` `/etc/motd`. No grandchildren, however, would inherit this ACL. This is an effective way to pass the use of an ACL from one program to another. You can use an inheritance level of `'-1'` to denote infinite inheritance. Naturally you should set the inheritance level to the minimum to allow the functionality needed.

Inheritance works for both File and Capability ACLs; however, it is most commonly needed for File ACLs.

Time-Based ACLs

Sometimes you may wish you could allow a program to have a special ACL only when it needs it. One of the most common of these situations is when you wish to rotate logs. If you gave the log rotation script the ability to write to the logs directory, an attacker who had gained root access could use it to rotate important logs out of existence to hide her activities. (I personally don't use log rotation scripts that delete logs, I prefer to keep them around forever - hard drive space be damned -- until I manually remove them after archiving.)

So we want to give our log rotation script, `/usr/sbin/logrotate`, the ability to `WRITE` to `/var/log` only when called from cron at the appropriate time. Newer versions of LIDS have the ability to set time restrictions for this ACL, so you'd set up your ACLs as follows:

```
lfs# crontab -l | grep logrotate
```

```
18 0 * * * * /usr/sbin/logrotate

# protect /var/log
lfs# lidsadm -A -o /var/log -j APPEND

# Exception for cron
lfs# lidsadm -A -s /usr/sbin/cron -o /var/log \
    -t 0018-0019 -i 2 -j WRITE
```

The above commands give cron WRITE access to /var/log during the window of time when logrotate will be called (00:18) with an inherit level of 2 (to catch logrotate and it's children). This will make sure that logrotate is only able to affect /var/log when called by cron. Had we given logrotate the WRITE access, an attacker could manually run it sufficient times to rotate the logs to /dev/null during that one minute window.

If you are using time-based ACLs, it is important to make sure you crontabs (/var/spool/cron/crontabs, /etc/cron.*) are protected as well, lest an attacker learn what times ACLs may be used to her benefit, or change those crontabs to run her own commands during the vulnerable window.

Conclusion

This concludes the third installment in this series. The next, and final, installment in this series will explore LIDS log entries, using Ptrace, some troublesome issues with LIDS and a brief discussion of what users should be using LIDS to protect.

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, SecurityFocus

[An Overview of LIDS, Part Two](#)

Brian Hatch, SecurityFocus

[Privacy Statement](#)

Copyright 2006, SecurityFocus