Secure Linux containers cookbook

Strengthen lightweight containers with SELinux and Smack

Skill Level: Advanced

Serge E. Hallyn (sergeh@us.ibm.com)
Advisory Software Engineer
IBM

03 Feb 2009

Lightweight containers, otherwise known as Virtual Private Servers (VPS) or Jails, are often thought of as a security tools designed to confine untrusted applications or users; but as presently constructed, these containers do not provide adequate security guarantees. By strengthening these containers using SELinux or Smack policy, a much more secure container can be implemented in Linux®. This article shows you how to create a more secure Linux-Security-Modules-protected container. Both the SELinux and Smack policy are considered works in progress, to be improved upon with help from their respective communities.

A common response when someone first hears about containers is "How do I create a secure container?" This article answers that question by showing you how to use Linux Security Modules (LSM) to improve the security of containers. In particular, it shows you how to specify a security goal and meet it with both the Smack and SELinux security modules.

For background reading on Linux Containers, see "LXC: Linux container tools" (developerWorks, February 2009).

Linux containers are really a conceptual artifice built atop several Linux technologies:

- *Resource namespaces* allow the manipulation of lookups of processes, files, SYSV IPC resources, network interfaces, and more, all inside of containers.

- *Control groups* allow resource limits to be placed on containers.
- *Capability bounding sets* limit the privilege available to containers.

These technologies must be coordinated in order to provide the illusion of containers. Two projects already provide this functionality:

- Libvirt is a large project that can create virtual machines using the Xen hypervisor, qemu emulator, and kvm, and also using lightweight containers.
- Liblxc is a smaller set of libraries and userspace commands written in part to help kernel developers quickly and easily test the containers functionality.

Because "LXC: Linux container tools" was written using liblxc as its foundation, I will continue with liblxc here; however, anything we do here can just as easily be done using libvirt's container support.

Major player 1: LSM

Before we start, if you know little about the LSM, here is a quick review. According to the Wikipedia entry: *Linux Security Modules (LSM) is a framework that allows the Linux kernel to support a variety of computer security models while avoiding favoritism toward any single security implementation. The framework is licensed under the terms of the GNU General Public License and is standard part of the Linux kernel since Linux 2.6.... LSM was designed to provide the specific needs of everything needed to successfully implement a mandatory access control module, while imposing the fewest possible changes to the Linux kernel. LSM avoids the approach of system call interposition as used in Systrace because it does not scale to multiprocessor kernels and is subject to TOCTTOU (race) attacks. Instead, LSM inserts "hooks" (upcalls to the module) at every point in the kernel where a user-level system call is about to result in access to an important internal kernel object such as inodes and task control blocks.... The project is narrowly scoped to solve the problem of access control to avoid imposing a large and complex change patch on the mainstream kernel. It is not intended as a general "hook" or "upcall" mechanism, nor does it support virtualization.... LSM's access control goal is very closely related to the problem of system auditing, but is subtly different. Auditing requires that every attempt at access be recorded. LSM cannot deliver that, because it would require a great many more hooks, so as to detect cases where the kernel "short circuits" failing system calls and returns an error code before getting near significant objects.*

System security consists of two somewhat contradictory goals. The first is to achieve complete and fine-grained access control. At every point that information can be leaked or corrupted, you must be able to exert control. Controls that are too coarse

is the same as being uncontrolled. For instance, if (at the extreme) all files must be classified as one type and any one file must be world-readable, then all files must be world-readable.

On the other hand, configuration must also be simple, otherwise administrators will often default to giving too much access (and I can't emphasize this enough -- this is the same as being uncontrolled). For instance, if making a program work requires thousands of access rules, then chances are an admin will give the program too many access rights rather than testing whether each access rule was really needed.

The two primary security modules in Linux each take a different view on how to handle this balance.

- SELinux begins by controlling everything while using an impressive policy language to simplify policy management.
- Smack is primarily concerned with providing a simple access control.

Major player 2: SELinux

SELinux is by far the most well-known MAC system for Linux (*mandatory access control*). While it certainly still has its detractors, the fact that the popular Fedora® distribution has been deployed with SELinux enforcing for years is a tremendous testament to its success.

SELinux is configured using a modular policy language which allows an installed policy to be easily updated by users. The language also provides interfaces, allowing more high-level statements to be used to represent a collection of low-level "allow" statements.

In this article, we will be using a new interface to define containers. While the interface itself will be quite large due to the many access rights you must give the container, using the interface to create a new container will be very simple. Hopefully the interface can become a part of the core distributed policy.

Major player 3: Smack

Smack is the Simplified Mandatory Access Control Kernel. It begins by labeling all processes, files, and network traffic with simple text labels. Newly created files are created with the label of the creating process. A few default types always exist with clearly defined access rules. A process can always read and write objects of the same label. Privilege to bypass the Smack access rules are controlled using POSIX capabilities, so a task carrying `CAP_MAC_OVERRIDE` can override the rules; a task carrying `CAP_MAC_ADMIN` can change the rules and labels. "POSIX file capabilities:

Parceling the power of root" (Resources) demonstrates these privileges.

Our security goal

Instead of simply blindly applying policy and hoping to end up with something useful, let's begin by defining a clear security goal. The simplicity of Smack actually limits the goals we can achieve, but we'll pursue the following goal:

1. Create containers with segregated file systems providing Web and ssh services.
2. Containers will be protected from each other. A container designated *vs1* cannot read files owned by another container *vs2* or kill its tasks.
3. The host can protect its key files from containers.
4. The outside world can reach the Web servers and ssh servers on the containers.

The general setup

In this article we'll do two experiments -- first we'll set up containers protected by SELinux, then containers protected by Smack. The experiments will share much of the preliminary setup.

You can use a real machine to do these experiments, but you may find it easier or more comforting to use a virtual machine. To use qemu or kvm, you can create a hard disk using `qemu-img create vm.img 10G`.

Boot the virtual machine from CDROM using a command like `kvm -hda vm.img -cdrom cdrom.iso -boot d -m 512M`. A good choice for a CDROM image is to go to fedoraproject.org/get-fedora and download an installation DVD for Fedora 10 for i386. Substitute the filename you download for `cdrom.iso` in the previous command. You can mostly use the installation defaults, but make sure to unselect *office and productivity* and select *software development*. You'll also want to install the bridge-utils, debootstrap, and ncurses-devel rpms, probably using the yum package manager.

Now you need to compile a custom kernel. Download the kernel-sources rpm, patch it with enable-netns.patch (see the Download section) to provide network namespaces (which will be upstream as of 2.6.29 but not in Fedora 10), change the configuration, then complete the compilation and installation, by following the following instructions as root:

```
yumdownloader --source kernel
rpm -i kernel*
cd rpmbuild
rpmbuild -bc SPECS/kernel-*
cd BUILD/kernel-2.6.27/linux-2.6*
patch -p1 < ~/enable-netns.patch
make menuconfig
make && make modules_install && make install
```

For both experiments, in the `make menuconfig` step, select *Network Namespaces* (under *Networking support -> Networking options* menus). For the Smack experiment, also go into the *Security options* menu, deselect *SELinux*, and select the next option, *Smack*. You may also need to switch the *default* boot entry in /boot/grub/grub.conf back to 0 instead of 1.

Now we want to try out liblxc. "LXC: Linux container tools" describes the basic usage of liblxc in detail, so we'll gloss over it here. Simply use the container_setup.sh script (see the Download section) to set up the bridge on which container network devices will talk. It will also clear your firewall, which by default isn't set up to handle the bridge, as well as set up the Smack policy (which we'll create later in the file /etc/smackaccesses) if you are doing the Smack experiment. You'll need to run container_setup.sh after each reboot or if you know how, make it run at boot automatically.

Now your machine is ready! Let's try out liblxc. You can download the latest source using cvs from lxc.sf.net and compile it using the following:

```
cvs -d:pserver:anonymous@lxc.cvs.sourceforge.net:/cvsroot/lxc
login
cvs -z3
-d:pserver:anonymous@lxc.cvs.sourceforge.net:/cvsroot/lxc co -P
lxc
cd lxc
./bootstrap && ./configure && make && make install
```

Now if you look at the README, you'll see there are quite a few options for getting started. Containers can be extremely lightweight because they can share many resources with your system -- including the filesystem. But our goal is to provide some simple isolation so we will use the script lxc-debian to create a full debian chroot image for each container. Begin by creating a container named `vsplain`:

```
mkdir /vsplain
cd /vsplain
lxc-debian create
        container name: vsplain
        hostname: vsplain
        IP 10.0.2.20
        gateway: 10.0.2.2
```

The configuration for this container is stored under the directory

/usr/local/var/lxc/vsplain. If you look at the file called cgroup, you'll see some lines beginning with `devices.`. These are directives to the devices whitelist cgroup which will mediate device creation, read, and write by the container.

Start this container using the command `lxc-start -n vsplain`. You'll be presented with a login prompt. Login to the container using username root with no password. Finally, when your container is up and running, you will want to

```
apt-get install openssh-server
apt-get install apache
```

Now you can ssh from the kvm host to the container and look at its Web page using 10.0.2.20 for vsplain's ip address and 10.0.2.15 for the host's. You can shut the container down at any time from a root terminal on the kvm host using the command `lxc-stop -n vsplain`.

At this point, you may want to save yourself some time by cloning two new virtual machines from this template. Shut down your vm and do:

```
cp vm.img selinux.img
cp vm.img smack.img
```

SELinux-protected containers

The SELinux policy for containers we'll use will consist of a policy module; the module has been posted to *refpolicy -- SELinux Reference Policy development mail list*. Download the policy into a directory /root/vs, into files called vs.if, vs.fc, and vs.te respectively. Compile and install the new module as follows:

```
cp -r /usr/share/selinux/devel /usr/share/selinux/vs
cp /root/vs.?? /usr/share/selinux/vs/
cd /usr/share/selinux/vs
make && semodule -i vs.pp
```

Then create containers /vs1 and /vs2 using lxc-debian and relabel their filesystems using

```
mkdir /vs1; cd /vs1
lxc-debian create
        container name: vs1
        hostname: vs1
        address: 10.0.2.21
        gateway: 10.0.2.2
        arch: 2 (i386)
mkdir /vs2; cd /vs2
lxc-debian create
```

```
        container name: vs2
        hostname: vs2
        address: 10.0.2.22
        gateway: 10.0.2.2
        arch: 2 (i386)
fixfiles relabel /vs1
fixfiles relabel /vs2
```

When you start your containers (for instance by using `lxc-start -n vs1`), you'll likely get a few audit messages about SELinux access denials. Don't worry -- the container starts up fine with network services enabled *and* the containers are now isolated. If you help container vs1 cheat using `mount --bind / /vs1/rootfs.vs1/mnt` before starting the container, you'll find that even though you are the root user, ls /mnt/root will be refused.

To see how this works, let's look at the vs.if interface file. This defines an interface called `container` which takes one argument, the base name for the container to define. The vs.te file calls this function twice with the container names vs1, vs2. In the interface, `$1` is expanded to the argument, so `$1_t` becomes `vs1_t` when we call `container(vs1)`. (From here on let's assume we are defining vs1).

The most important lines are those involving `vs1_exec_t`. The container runs in type `vs1_t`. It enters this type when `unconfined_t` executes the container's /sbin/init which is of type `vs1_exec_t`.

Most of the rest of the policy merely is there to grant the container sufficient privilege to access bits of the system: network ports, devices, consoles, etc. The interface is as long as it is due to the fine-grained nature of the existing SELinux reference policy. As we're about to see, the Smack-protected container will have a much simpler policy; in return, it will promise much less flexible protection from misbehaving system services.

There is one more thing you need to do. You may have noted that while the container is not able to overwrite its `$1_exec_t`, that is /sbin/init. But what it can do is something like

```
mv /sbin /sbin.bak
mkdir /sbin
touch /sbin/init
```

The resulting /sbin/init will be of type `vs1_file_t`. Why do you think the container admin would want to do this? Because it would launch the container, including the ssh daemon, in the `unconfined_t` domain, giving him a privileged shell and allowing him to escape the SELinux constraints we were trying to enforce.

To prevent this, you actually want to start the container through a custom script and relabel sbin/init to `vs1_exec_t` before starting the container. In fact, you can copy a pristine copy of init back into the container and relabel that if the container

administrator didn't mind. But we'll just relabel the existing init:

```
cat >> /vs1/vs1.sh << EOF
#!/bin/sh
chcon -t vs1_exec_t /vs1/rootfs.vs1/sbin/init
lxc-start -n vs1
EOF
chmod u+x /vs1/vs1.sh
```

Now you'll need to start the container using `/vs1/vs1.sh` instead of using `lxc-start` by hand.

Smack-protected containers

Recompile the kernel with Smack enabled. You should be able to simply enter the `/root/rpmbuild/BUILD/kernel*/linux*` directory, `make menuconfig`, go to the *security* menu, disable SELinux, and enable Smack. Then just repeat the steps `make && make modules_install && make install`.

Also stop userspace from trying to configure SELinux. You can do this through the SELinux administration GUI or you can edit /etc/selinux/config and set `SELINUX=disabled`. You also will want to do a few more steps to install a Smack policy at boot:

```
mkdir /smack
cd /usr/src
wget http://schaufler-ca.com/data/080616/smack-util-0.1.tar
tar xf smack-util-0.1.tar; cd smack-util-0.1
make && cp smackload /bin
```

The actual Smack policy looks like Listing 1:

Listing 1. smackaccesses

```
vs1 _ rwa
_ vs1 rwa
vs2 _ rwa
_ vs2 rwa
_ host rwax
host _ rwax
```

It should be copied into a file called /etc/smackaccesses. The next time you run /bin/container_setup.sh, it will load this file into `smackload`.

The policy is pretty simple. By default, any label can read data labeled _. We define a new label `host` for host-private data which containers should not be able to access; we assign this to the cgroups filesystem in the container_setup.sh script.

Other sensitive files like /etc/shadow should certainly get this label.

We define `vs1` and `vs2` to label containers. By default they can each access their own data. We add a rule to allow them to write _ so as to allow sending network packets. Since `vs1` cannot access `vs2` data and vice versa, containers are protected from each other.

As mentioned before, the ability to define or bypass Smack access rules is determined by the `CAP_MAC_ADMIN` and `CAP_MAC_OVERRIDE` capabilities. So you will need to keep containers from having those capabilities. You can do that using the a helper program dropmacadmin.c (in the Download section). You must compile it statically since the containers have different library versions from the host:

```
gcc -o dropmacadmin dropmacadmin.c -static
cp dropmacadmin /bin/
```

Create a new container called `vs1`:

```
mkdir /vs1; cd /vs1
lxc-debian create
        container name: vs1
        hostname: vs1
        address: 10.0.2.21
        router: 10.0.2.2
        arch: 2 (i386)
```

Label all files in `vs1`'s filesystem with the label `vs1`:

```
for f in `find /vs1/rootfs.vs1`; do
        attr -S -s SMACK64 -V vs1 $f
done
```

Now you need to create a script which will start the container safely. What this means is that it will set it's process label to `vs1` and wrap the container's /sbin/init through dropmacadmin (like so):

```
cat >> /vs1/vs1.sh << EOF
#!/bin/sh
cp /bin/dropmacadmin /vs1/rootfs.vs1/bin/
attr -S -s SMACK64 -V vs1 /vs1/rootfs.vs1/bin/dropmacadmin
echo vs1 > /proc/self/attr/current
lxc-start -n vs1 /bin/dropmacadmin /sbin/init
EOF
chmod u+x /vs1/vs1.sh
```

One more thing will let `vs1` write to the tmpfs filesystem it is going to mount:

```
sed -i 's/defaults/defaults,smackfsroot=vs1,smackfsdef=vs1/' \
        /vs1/rootfs.vs1/etc/fstab
```

This will cause the tmpfs filesystem mounted at /dev/shm to carry the `vs1` label so that `vs1` can write to it. Otherwise, `vs1` init scripts won't be able to create the /dev/shm/network directory it uses while setting up the network. Similarly, if you want to use a ram-based /tmp, you'll want those same options.

Now again let's help `vs1` cheat. Create `vs2` the same way you created `vs1`, substituting `vs2` for `vs1` at each step. Then bind-mount the root filesystem under `vs1`'s /mnt:

```
mount --bind /vs1 /vs1
mount --make-runbindable /vs1
mount --rbind / /vs1/rootfs.vs1/mnt
```

Start the container using vs1.sh. Note that you can still see the Web page on `vs1` and `vs2` from the kvm host. Note also that `vs1` cannot access `vs2` over the network. It also can't look through `vs2`'s files:

```
vs1:~# ls /mnt/
  (directory listing)
vs1:~# ls /mnt/vs2/rootfs.vs2
  ls:/mnt/vs2/rootfs.vs2: Permission denied
vs1:~# mkdir /cgroup
vs1:~# mount -t cgroup cgroup /cgroup
vs1:~# ls /cgroup
  ls:/mnt/vs3: Permission denied
vs1:~# mknod /dev/sda1 b 8 1
  mknod: `/dev/sda1': Operation not permitted
vs1:~# mount /mnt/dev/sda1 /tmp
  mount: permission denied
```

It can look through the host file system. Anything we want to protect against, we can label with the `host` label. That's what we did with the cgroup filesystem which is why `ls /cgroup` failed.

Finally, the devices whitelist cgroup is preventing us from creating a disk device, as well as mounting it if it exists (as it does through /mnt).

Of course, the way we've set this up, the container admin can *remove* `/mnt/dev/sda1`, as well mess up the host in any number of ways, so other than as demonstration this bind mount is obviously not desirable!

Note that while on the SELinux system, the default (and easy) route was to allow the containers to talk to each other over the network, the inverse is true in Smack. Allowing containers to talk to each other is currently very hard to do. An ability to set labels on IP addresses is coming soon though and should allow us to set up policy to allow containers to communicate.

Related to how we set up Smack networking, we have another problem. The command `kill -9 -1` kills every task on the system. When done by a task in a container, this should only kill tasks in the same container. That behavior is now fixed in the upstream kernel, but not in the Fedora 10 kernel we are using. So every task will be sent a `-9` signal.

In the SELinux-protected containers, SELinux stops the signals from passing the container boundary, so `kill -9 -1` is actually safe. But in Smack tasks by default are labeled _ just as the network is, so since we allowed the container to write _ to allow writing to the network, and since killing a task is considered a write access by Smack, you are also allowing the container admin to kill any tasks on the whole system.

Another shortcoming (which is also present in the SELinux containers) has to do with Unix98 pseudo-terminals. Open two graphical terminals. In the first, start up vs1 and look under `/dev/pts`. You will see at least two entries, 0 and 1, one belonging to each terminal. From the vs1 container you are able to write into the entry corresponding to the other terminal.

With the Fedora kernel there are two solutions. You can use the device whitelist cgroup to deny the container the ability to open the devices. However, this will have to be done by hand each time the container is started in order to grant it access to its terminal; or you can achieve the same effect by applying SELinux and Smack labels.

The newer 2.6.29 kernel supports devpts namespaces. A container will remount /dev/pts, after which it will be unable to access the devpts entries belonging to the host or other containers.

Conclusion

This article showcased the basic tools for creating LSM-protected containers, but much work remains to be done:

- For Smack, you must choose files to label as `host`.
- For SELinux, you should fine-tune and then push a `container` interface into the upstream reference policy.

While such work is ongoing, and until more experience is gained with LSM-protected containers, you should not put all your trust in these mechanisms to protect against an untrusted root user.

Although there are no established best practices for creating containers yet (that I know of), there are a few ideas worth starting with. First, remember you are consolidating two somewhat contradictory goals: You want to minimize duplication

among containers (and the host) while needing to ensure isolation. One way to achieve these goals could be to create a single full minimal rootfs in which no container runs and labeling it a type which all containers can read. Then use a custom version of the lxc-sshd script to create each actual container based on the prototype, creating read-only mounts for most of the container's filesystem while providing a private writable place for the container to store files, say like /scratch. Since each container has a private mounts namespace, it can bind-mount any files or directories which it needs to be private and/or writeable from its private shared directory. For instance, if it wants a private /lib, it can `mount --bind /scratch/rootfs/lib /lib`. Likewise, the admin can ensure that every container does `mount --bind /scratch/shadow /etc/shadow` at startup.

One clear limitation of the approach I demonstrated here with both SELinux and Smack is that the container administrator cannot exploit LSM to control information flow within his own container. Rather, for simplicity, all tasks in the container are treated the same by MAC policy. In another article, I hope to explore how to allow container administrators to specify their own LSM policies without allowing them to escape their own contraints.

Acknowledgments

Casey Schaufler, the author of Smack, helped in getting the Smack-protected container off the ground, and Dan Walsh was kind enough to provide feedback on the SELinux policy.

Downloads

Description	Name	Size	Download method
Code for this article	code.zip	3KB	HTTP

Information about download methods

More downloads

- Demo: SELinux containers policy

Resources

Learn

- "LXC: Linux container tools" (developerWorks, February 2009) is a step-by-step guide to creating Linux containers.
- "POSIX file capabilities: Parceling the power of root" (developerWorks, October 2007) showcases the Linux POSIX file capabilities that split root user powers into smaller privileges.
- The refpolicy -- SELinux Reference Policy development mail list is where you'll find the policy module we used in this article.
- "SELinux from scratch" (developerWorks, May 2006) is a detailed introduction to SELinux.
- Planet SELinux is an aggregation of blog posts from members of the SELinux development community.
- "Reference Policy for Security Enhanced Linux" (PDF document) is a paper presenting the SELinux reference policy.
- "Smack for simplified access control" (LWN.net, August 2007) is an early writeup on the Smack submission.
- Other container technologies include
  - Solaris Zones (Solaris)
  - BSD jails (FreeBSD)
  - Linux-Vserver (Linux)
  - OpenVZ (Linux)
  - FreeVPS (Linux)
- In the developerWorks Linux zone, find more resources for Linux developers (including developers who are new to Linux), and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Linux Resource Containers project on SourceForge.net is a repository of code for application container implementation in the Linux kernel, a staging area for code that may be sent to the linux-kernel mailing list.

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and spaces.

About the author

Serge E. Hallyn
Serge Hallyn is a part of IBM's Linux Technology Center, focusing on Linux kernel and security. He obtained his Ph.D. in computer science from the College of William and Mary. He has written and contributed to several security modules. He currently focuses on adding support for virtual server functionality, application checkpoint/restart, and POSIX file capabilities.

Trademarks