Partitioning a Server with NSA SE Linux

Russell Coker <russell@coker.com.au>,
http://www.coker.com.au/

Abstract

The requirement to purchase multiple machines is often driven by the need to have multiple administrators with root access who do not trust each other.

Having large numbers of expensive under-utilised servers with the associated management costs is not ideal.

I will describe my solution to this problem using SE Linux [1] to partition a server such that the "root" users can't access each other's files, kill each other's processes, change passwords for each other's users, etc.

DOS attacks will still be possible by excessive use of memory and CPU time, but apart from that all the benefits of separate hardware will be provided.

1 Introduction

SE Linux dramatically improves the security of a Linux system by adding another layer of security in addition to the default Unix permissions model. This is accomplished by firstly assigning a *type* to every file, device, network socket, etc. Then every process has a *domain*, and the level of access permitted to a type is determined by the domain of the process that is attempting the access (in addition to the usual Unix permission checks). Domains may only be changed at process execution time. The domain may automatically be changed when a process is executed based on the type of the executable program file and the domain of the process that is executing it, or a privileged process may specify the new domain for the child process.

In addition to the use of domains and types for access control SE Linux tracks the *identity* of the user (which will be *system_u* for processes that are part of the operating system or the Unix username) and the role. Each *identity* will have a list of roles that it is permitted to assume, and each *role* will have a list of domains that it may use. This gives a high level of control over the actions of a user which is tracked through the system. However in this paper I am not using any of the identity or role features of SE Linux (they merely give an extra layer of security on top of what I am researching). So I will not mention them again.

For a detailed description of how SE Linux works I recommend reading the paper Peter Loscocco presented at OLS in 2001 [1]. For the details of SE Linux policy configuration I recommend Stephen Smalley's paper [2].

The problem of dividing a hardware resource that is expensive to purchase and manage among many users is an ongoing issue for system administrators. In an ISP hosting environment or a software development environment a common method is to use *chroot* environments to partition a server into different virtual environments. I have devised some security policies and security server programs to implement a chroot environment on SE Linux with advanced security offering the following features:

1. An unpriviledged user (chroot administrator) can setup their own chroot environment, create accounts, run sshd, let ordinary users login via ssh, and do normal root things to them without needing any help from the administrator.

2. Processes outside the chroot environment can see the processes in the chroot, kill them, strace them, etc. Also the user can change the security labels of files in their chroot to determine which of the files are writable by a process in the chroot environment (this can be

done from outside the chroot environment or from a priviledged process within the chroot environment).

3. Processes inside the chroot can't see any system processes, or any of the user's processes that are outside the chroot. They can see which PID's are in use, but that doesn't allow them to see any information on the processes in question or molest such processes. The chroot processes won't be allowed to read any files or directories labelled with the type of the user's home directory. This is so that if the wrong files are accidentally moved into the chroot environment then they won't be read by chroot processes.

4. The administrator can set a user's account for multiple independant chroot environments. In such a case processes inside one chroot can't interfere with processes in other in any way, and all their files will be in different types. This prohibits a nested chroot, but I think that there's no good cause for nested chroot's anyway. If someone has a real need for nested chroot's they could always build them on top of my work - it would not be particularly difficult.

5. The user will have the option of running a priviledged process inside their chroot which can do anything to files or processes in the chroot (even files that are read-only to regular chroot processes) but which can't do anything outside the chroot. Also if this process runs a program that is writable by a regular chroot process then it runs it in the regular chroot process domain. This is to prevent a hostile user inside the chroot from attacking the chroot administrator after tricking them into running a compromised binary.

A basic chroot environment has several limitations, the most important of which is that a process in the chroot environment can kill any process with the same UID (or in the case of a root process any process with any UID) outside the chroot environment. The second major limitation is that root access is needed to call the *chroot()* system call to start a chroot environment and this gives access to almost everything else that you might want to restrict. There are a number of other limitations of chroot environments, but they are all trivial compared to these.

One method of addressing this issue is that of GR security [3]. GR Security locks down a chroot environment tightly, preventing fchdir(), mount(), double-chrooting, pivot_root(), and access to non-chroot processes (or processes in a different chroot). It also has the benefit that no extra application configuration is required. However it has the limitation that you have very limited ability to configure the capabilities of the chroot, and it has no solution to the problem of requiring root access to setup the chroot environment.

Another possible solution to the problem is that of BSD Jails [4]. A jail is a chroot environment that prevents escape and also prevents processes in the jail from interfering with processes outside the jail. It is designed for running network servers and allows confining the processes in a particular jail to a single IP address (a very handy feature that is not available in SE Linux at this time and which I have to emulate in a shared object). Also the jail authors are working on a *jailinit* program which is similar to *init* but for jails (presumably the main difference is that it doesn't expect to have PID==1), this program should work well for chroot environments in SE Linux too.

Another possible solution is to use User-Mode Linux [5]. UML is a port of the Linux kernel to the Linux system call interface. So it can run a kernel as a user (UID!=0) application using regular files on the file system for storage. One problem with UML is that it is based around file system images on disk, so to access them without the UML kernel running you have to loopback mount them which is inconvenient. Also starting or stopping a UML image is equivalent to booting or shutting down a server (which is inconvenient if you just want to run a simple command like *last*). A final problem with UML for hosting is that using file systems for storage is inefficient, if you have 100 users who each might use 1G of storage (but who use on average 50M) you need 100G of disk space. With chroot based systems you would only need 5G of disk space. Finally backing up a file system image from a live UML setup will give a corrupted file system, backing up files from a chroot is quite safe.

2 Isolation of the Chroot

The first aim is to have the chroot environment be as isolated as possible from other chroot's, from the chroot administrator, and from the rest of the system. This is quite easy as SE Linux defaults to denying access (apart from the system administration domain *sysadm_t* which has full access to see and kill processes, read files, etc).

For example pivot_root, is denied by not giving the *sys_admin* capability. Access to other processes is denied by not permitting access to read from the /proc directories, send signals, ptrace, etc. *fchdir()* is prevented because processes in the chroot don't get access to files or directories outside the chroot because of the type labels, so it can only fchdir() back into the chroot! Mount and chroot accesses are also not granted to the chroot environment, nor is access to configure network interfaces.

The next aim is to make a chroot environment more secure than a non-SE system in full control of the machine. The first step to this is having critical resources such as hard drive block devices well out of reach of the chroot environment. With a default SE Linux security policy (which my work is based on) that is already achieved, a regular root process that is not chrooted will be unable to perform such access. This isolation of block devices, the boot manager, priviledged system processes (init, named, automount, and the mail server) from the rest of the system already makes a chroot environment on SE Linux more secure than that on a system without SE Linux, most (hopefully all) attacks against such critical parts of the system that would work on other systems will be defeated by SE Linux.

One problem that I still have not solved to my satisfaction is that of not requiring super-user access to initially create the chroot environment. I have developed policy modifications to allow someone to login as root (UID==0) in a non-chroot environment and not have access to cause any damage. So for my current testing I have started all chroot environments from a root login in a user domain as the *chroot* and *mount* operations require root privileges. For production use you would probably not want to give a user-domain root account to a user, so I am writing a SUID root program for running both *mount* and *chroot* to start a chroot environment and to run umount to end such an environment (after killing all the processes). It will take a configuration file listing the target directory, the bind mounts to setup, and the programs to run. One issue that I initially had with this was to make sure that it only runs on SE Linux (if SE Linux was removed but the application remained installed then you would grant everyone the ability to run programs as root in an unrestricted fashion through chrooting to the root directory). My current method of solving this is to execute */bin/false* at the start of the program, if SE Linux blocks that execution then it indicates that SE Linux is in enforcing mode (otherwise the program ends). Also it directly checks for the presence of SE Linux (so if the administrator removes */bin/false* then it won't give a false positive). However this still leaves the corner case where an administrator puts the machine in permissive mode and removes */bin/false*. To avoid this the program will then try reading */etc/passwd* which will always be readable on a functional Unix machine unless you have SE Linux (or some similar mechanism) in place. Before running *chroot* the wrapper will change directory to the chroot target and run "chroot .". My policy prohibits the *chroot* program from accessing any directories outside the chroot environment to prevent it from chrooting to the wrong directory, thus an absolute path will not work. Chroot administrators would be confused by this, so the wrapper hides it from them.

I believe that this method will work, but I have not tested it yet.

3 Domains

It would be possible to create a SE Linux policy to allow a chroot to have all the different domains that the full environment has (IE a separate security domain for each daemon). However this would be a huge amount of work for me to write the policy and maintain it as new daemons become available and as new versions of daemons are released with different functionality. It would result in a huge policy (number of daemons multiplied by number of chroot environments could result in a large number of domains) which is currently held in non-pageable kernel memory. In a future version of SE Linux they plan to make it pageable, but the fact that a large policy has a performance impact will remain. Also the typical chroot administrator does not want the bother of working with this level of complexity. Finally the limited scope of a chroot environment (no dhcp client, no need to run fsck or administer RAID devices, etc) reduces the number of interactions that can have a security impact, and as a general rule there is a security benefit in simplicity as mistakes are less likely.

In my current policy I have a single domain for the main programs in the chroot environment. This domain has write access to files under /home, /var, /tmp, and anywhere else that the chroot administrator desires (they can change the type of files and directories at any time to determine where programs can write, the writable locations of /home, /var, and /tmp are merely default values that I recommend - they are not requirements). Typically the entire chroot would default to being writable when it is installed and the chroot administrator would be encouraged to change that according to their own security needs. In the case of a chroot setup with *chroot(etbe, etbe_apache)* the main domain will be called *etbe_apache_t*.

I have been considered having a separate domain for user processes inside the chroot so that they can only write to files under /home (for example) and not /var (which would only be writable by system processes). To implement this would require either an automatic domain transition rule to transform the domain when running a user shell (as opposed to a shell script run by a system program), or a modified sshd, ftpd, etc. Requiring that chroot administrators use modified versions of standard programs is simply not practical, most users would be unable to manage correct installation and would require excessive help from the system administrator, also it might require significant programming to produce modified versions of some login programs. This leaves the option of automatic transitions. Doing this would require that a *chsh* inside the chroot not allow changing the shell to */bin/sh* or any other shell that would be used by system shell scripts, and that all shells listed in */etc/shells* be labelled as user shells. This requires a significant configuration difference between the chroot setup and that of standard configurations, that is difficult to maintain because of distribution packages and graphical system administration tools that would tend to change them back.

I conclude that having a separate domain for user processes is not feasible.

The next issue is administration of the chroot. Having certain directories and files be read-only is great for security but a pain when it is time to upgrade! This is a common problem for administrators who use a read-only mount for their chroot environment. To solve this issue I have created a domain which has write privileges for all files in the chroot, for the example above the name of this domain would be *etbe_apache_super_t*. This domain also has control over all processes in the chroot (ability to see them, kill them, and ptrace them), while the processes in the chroot can not even determine the existance of such super processes. If this process executes a file that is writable by the main chroot domain then it will transition to the main chroot domain, so that if a trojan is run it will not be able to damage the read-only files. This protection is not perfect however, using the *source* command or `/bin/sh < script` to execute a shell script will result in it being run in the super domain. However I think that this is a small problem, tricking an administrator into redirecting input for a shell from a hostile script or using the source command is very difficult (but not impossible). However it is quite easy for an administrator to mistakenly give the wrong label to files and allow them to be written by the wrong people (I made exactly this mistake when I first set it up), so preventing the execution of binaries that may have been compromised is a good security measure.

Note, that this method of managing read-only files does not require that applications running in the chroot environment be stopped for an upgrade, unlike most other methods of denying write access to files that are in use.

To have a working chroot environment you need a number of device files to be present, pseudo-tty files (for logins and expect), /dev/random (for sshd), and others. The chroot administrator can not be permitted to create their own device nodes as this would allow them to create /dev/hda and change system data! So I have created a special mount domain which in this example would be named etbe_mount_t based on Brian May's mount policy to allow bind mounts of these device nodes. This does have one potential drawback, if you are denying a domain access to device nodes solely by denying search access to /dev then bind mounts could be used to allow access in contravention of security policy! I could imagine a situation where someone would want to allow a domain to access the pseudo-tty it inherits from it's parent but not open other pty's of the same type, and implementing this by denying access to the /dev/pts. This is not something that I would recommend however. I believe that this is the best solution, as not having a user mount domain would require that either the system administrator create bind mounts (a process that has risks regarding sym-links etc), create special device nodes (having two different device nodes for the same device with different types is a security issue), or otherwise be involved in the setup of the chroot in a fashion that involves work and security risks.

In summary, if you have the bad idea of restricting access to the /dev directory to prevent access to device nodes then things will break for you, however there are many other ways of breaking such things so I think that the net result is that security will not be weakened.

To actually enter the chroot environment you need to execute the *chroot()* system call. To allow that I created a domain for chroot, which in this example will be called *etbe_chroot_t*, this domain is entered when the user domain *etbe_t* executes the chroot program. Then when this domain executes a file of type *etbe_apache_ro_t* or *etbe_apache_rw_t* it will transition to domain *etbe_apache_t*, and when it executes a file of type *etbe_apache_super_entry_t* it will transition to domain *etbe_apache_super_t*.

4 Types

The primary type used for labelling files and directories is for read/write files, in the case of a chroot setup by *chroot(etbe, etbe_apache)* the type will be *etbe_apache_rw_t*. It can be read and written by *etbe_apache_super_t* and *etbe_apache_t* domains, and read by the *etbe_chroot_t* and *etbe_mount_t* domains. It can also be read and written by the user domain *etbe_t*.

The type *etbe_apache_ro_t* is the same but can only be written by the user domain and *etbe_apache_super_t*.

To enter as domain *etbe_apache_super_t* I have defined a type *etbe_apache_super_entry_t*. The aim of this is to allow easy entry of the administration domain by a script, otherwise I might have chosen to have a wrapper program to enter the domain which prompts the user for which domain they want to enter. The wrapper program idea would have the advantage of making it easier for novice chroot administrators, and I may eventually implement that too so that users get a choice.

One problem I found when I initially setup a chroot was that when installing new Debian packages the post installation scripts (running in the *etbe_apache_super_t* domain) started a daemon (such as sshd) it would also start as *etbe_apache_super_t*, and then a user could login with that domain! To solve this problem I created a new type *etbe_apache_dropdown_t*, when the *etbe_apache_super_t* executes a program of that type it transitions to *etbe_apache_t*, so labelling the */etc/init.d* directory (and all files it contains) with this type causes the daemons to be executed in the correct domain. The write access for this type is the same as that for etbe_apache_ro_t.

5 Configuring the Policy

I have based my policy for chroot environments around a single macro that takes two parameters, the name of a user domain, and the name of a chroot. For example if I have a etbe_t domain and I want to create a chroot environment for apache then I could call *chroot(etbe, etbe_apache)* to create the chroot.

This makes it convenient to setup for a basic chroot as all the most likely operations that don't comprise a security risk are allowed. Naturally if you have different aims then you will sometimes need to write some more policy. One of my machines runs a chroot environment for the purpose of running Apache, it required an extra fourteen lines of SE policy configuration to allow the Apache log files to be accessed by other system processes outside the chroot (a script that runs a web log analysis program in particular).

The typical chroot environment used for an Internet server will probably require between 5 and 20 lines of extra policy configuration and will take an experienced administrator less than 30 minutes to setup. Of course these could be added to custom macros allowing bulk creation with ease, setting up 500 different chroot environments for Apache should not take more than an hour!

Here is my policy for an Apache web server run in the system_r role by the system init scripts that has read-only access to all files under /home (which is

–bind mounted inside the chroot), and which allows logrotate to run the web log analysis scripts as a cron job.

```
# setup the chroot
chroot(initrc, apache_php4)
# allow apache to change UID/GID and to bind to the port
allow apache_php4_t self:capability { setuid setgid net_bind_service };
allow apache_php4_t http_port_t:tcp_socket name_bind;
allow apache_php4_t tmpfs_t:file { read write };
# allow apache to search the /home/user directories
allow apache_php4_t user_home_dir_type:dir search;
# allow apache to read files and directories under the users home dir
r_dir_file(apache_php4_t, user_home_type);
# allow logrotate to enter this chroot (and any other chroot environments
# that are started from initrc_t)
domain_auto_trans(logrotate_t, chroot_exec_t, initrc_chroot_t)
# this chroot is located under a users home directory so logrotate needs to
# search the home directory (x access in directory permissions) to get to it
allow logrotate_t user_home_dir_t:dir search;
# allow logrotate to search through read-only directories (does not need read
# access) and read the directories and files that the chroot can write (the
# web logs).  NB I do not need to restrict logrotate this much - but why give
# it more than it needs?
allow logrotate_t apache_php4_ro_t:dir search;
r_dir_file(logrotate_t, apache_php4_rw_t)
# allow a script in the chroot to write back to a pipe created by crond
allow initrc_chroot_t { crond_t system_crond_t }:fd use;
allow initrc_chroot_t { crond_t system_crond_t }:fifo_file { read write };
allow apache_php4_t { crond_t system_crond_t }:fd use;
allow apache_php4_t { crond_t system_crond_t }:fifo_file { read write };
```

That's 14 lines because I expanded it to make the policy clearer. Otherwise I would probably compress it to 10 lines.

6 Networking

The final issue is networking, the BSD Jail facility has good support for limiting network access via a single IP address per jail.

SE Linux lacks support for controlling networking in this fashion, server ports can only be limited by the port number. This is OK if different chroot environments provide different services. Also this works if you have an intelligent router in front of the server to direct traffic destined for different IP addresses to different ports on the same IP address (in which case the different chroot environments can be given permissions for different ports).

I have an idea for another solution to this problem which is more invasive of the chroot environment. The idea is to write a shared object to be listed in */etc/ld.so.preload* which will replace the *bind()* system call. This will then communicate over a Unix domain socket (which would be accessed through a bind mount) to a server process run with system privileges, and it will pass the file descriptor for the socket to it. The server process will then use the *accept_secure()* system call to determine the security context of the process that is attempting the bind, it will then examine some sort of configuration database and decide whether to allow the bind, or whether to modify it. If the bind parameters have to be modified (for example converting a bind to *INADDR_ANY* to be a bind to a particular IP address) then it would do so. Then it would do a *bind()* system call and return a success code to the socket that connects to the application.

This support would be ideal as it would be easiest to automate and would allow setting up hundreds or thousands of chroot environments at the same time with ease.

Unfortunately I couldn't get this code fully written in time for the publishing deadline.

7 Conclusion

I believe that I have achieved all my aims regarding secure development environments or other situations where there is no need to run network servers. The policy provides good security and allows easy management.

My current design for entering a chroot environment should work via a SUID root program, but I will have to test it. The current method of allowing unprivileged root logins has been tested in the field and found to work reasonably well.

Currently the only issue I have not solved to my satisfaction is that of binding chroot environments to specific IP addresses. Currently the best option that I have devised involves pre-loading a shared object into all processes in the chroot environment (which will inconveniance the user). But I have not yet implemented this so I am not certain that it will work correctly.

8 Obtaining the Source

Currently most of my packages and source are available at `http://www.coker.com.au/selinux/` however I plan to eventually get them all into Debian at which time I may remove that site.

I have several packages in the unstable distribution of Debian, the first is the *kernel-patch-2.4-lsm* and *kernel-patch-2.5-lsm* packages which supply the Linux Security Modules `http://lsm.immunix.org/` kernel patch. That patch includes SE Linux as well as LIDS and some of the OpenWall functionality. When I have time I back-port patches to older kernels and include new patches that the NSA has not officially released, so often my patches will provide more features than the official patches distributed by the NSA from `http://www.nsa.gov/selinux/index.html` or the patches distributed by Immunix. However if you want the *official* patches then these packages may not be what you desire.

From the *selinux-small* archive I create the packages *selinux* and *libselinux-dev* which are also in the unstable distribution of Debian.

9 Acknowledgments

Thanks to Dr. Brian May for reviewing this paper and providing advice.

References

[1] *Meeting Critical Security Objectives with Security-Enhanced Linux*
Peter A. Loscocco, NSA, loscocco@tycho.nsa.gov
Stephen D. Smalley, NAI Labs, ssmalley@nai.com
`http://www.nsa.gov/selinux/ottawa01-abs.html/`

[2] *Configuring the SELinux Policy*
Stephen D. Smalley, NAI Labs, ssmalley@nai.com
`http://www.nsa.gov/selinux/policy2-abs.html/`

[3] *GRSecurity Site*
Brad Spengler, spender@grsecurity.net
`http://www.grsecurity.org/papers.php/`

[4] *Jails: Confining the omnipotent root*
Poul-Henning Kamp, phk@FreeBSD.org

Robert N. M. Watson, rwatson@FreeBSD.org
`http://docs.freebsd.org/44doc/papers/jail/jail.html/`

[5] *User-Mode Linux*
Jeff Dike, jdike@karaya.com
`http://user-mode-linux.sourceforge.net/`