SE Debian: how to make NSA SE Linux work in a distribution

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

Abstract

I conservatively expect that tens of thousands of Debian users will be using NSA SE Linux [1] next year. I will explain how to make SE Linux work as part of a distribution, and be managable for the administrator.

Although I am writing about my work in developing SE Linux support for Debian, I am using generic terms as much as possible, as the same things need to be done for RPM based distributions.

1 Introduction

SE Linux offers significant benefits for security. It accomplishes this 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 user-name) 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. When the user runs SUID or SGID programs the original identity will still be tracked and their privileges in the SE security scheme will not change. This is very different to the standard Unix permissions where after a SUID program runs another SUID program it's impossible to determine who ran the original process. Also of note is the fact that operations that are denied by the security policy [2] have the *identity* of the process in question logged.

For a detailed description of how SE Linux works I recommend reading the paper Peter Loscocco presented at OLS in 2001 [1].

The difficulty is that this increase in functionality also involves an increase in complexity, and requires re-authenticating more often than on a regular Unix system (the SE Linux security policy requires that the user re-authenticate for change of *role*). Due to this most people who could benefit from SE Linux will find themselves unable to use it because of the difficulties of managing it. I plan to address this problem through packaging SE Linux for Debian.

The first issue is getting packages of software that is patched for support of the SE Linux system calls and logic. This includes modified programs for every method of login (/bin/login, sshd, and X login programs), modified *cron* to run cron jobs in the correct security context, modified *ps* to display the security context, modified *logrotate* to keep the correct context on log files, as well as many other modified utilities.

The next issue is to configure the system such that when a package of software is installed the correct security contexts will be automatically applied to all files.

The most difficult problem is ensuring that configuration scripts get run in the correct security context when installing and upgrading packages.

The final problem is managing the configuration files for the security policy.

Once these problems are solved there is still the issue of the SE Linux sample policy being far from the complete policy that is needed in a real network. I estimate that at least 500 new security policy files will need to be written before the sample policy is complete enough that most people can just select the parts that they need for a working system.

2 Patching the Packages

The task of the login program is to authenticate the user, *chown* the tty device to the correct UID, and change to the appropriate UID/GID before executing the user's shell. The SE patched version of the login program performs the same tasks, but in addition changes the security identifier (SID) on the terminal device with the *chsid* system call and then uses the *execve_secure* system call instead of the *execve* system call to change the SID of the child process. The login program also gives the user a choice of which of their authorised roles they will assume at login time.

This is not very different from the regular functionality of the login program and does not require a significant patch.

Typically this adds less than 9K to the object size of the login program, so hopefully soon many of the login programs will have the SE code always compiled in. For the rest we just need a set of packages containing the SE versions of the same programs. So this issue is not a difficult one to solve and most of the work needed to solve it has been done.

A similar patch needs to be applied to many other programs which perform similar operations. One example is *cron* which needs to be modified so cron jobs will be run in the correct security context. Another example is the *suexec* program from *Apache*. An example of a similar program for which no-one has yet written a patch is *procmail*.

Programs which copy files also need to have suitable options for preserving SIDs, *logrotate* and the *fileutils* package (which includes *cp*) have such patches, *cpio* lacks such a patch, and there is a patch for *tar* but it doesn't apply to recent versions and probably needs to be re-written.

3 Setting the Correct SID When Installing Files

When a package of software is installed the final part of the installation is running a *postinst* script which in the case of a daemon will usually start the daemon in question. However if the files in the package do not have the correct SIDs then the daemon may not be able to run, or will be unable to run correctly!

The Debian packaging system does not currently have any support for running a script after the files of a package are installed but before the *postinst* script. There have been discussions for a few years on how best to do this, as I didn't have time to properly re-write *dpkg* I instead did a quick hack to make it run scripts that it finds in */etc/dpkg/postinst.d/* before running the *postinst* of the package.

When installing an SE Linux system the program *setfiles* is used to apply the correct SIDs to all files in the system. I have written a patch to make it instead take a list of canonical fully-qualified file names on standard input if run with the *-s* switch, which is now included in the NSA source release.

The combination of the *dpkg* patch and the *setfiles* patch allow me to solve the basic problem of getting the correct SIDs applied to files, my script just queries the package management system for a list of files contained in the package and pipes it through

to *setfiles* to set the SID on each file.

The next complication is setting the correct SID for the *setfiles* program, by default it gets installed with the security type *sbin_t* because that is the type of the directory it is installed in. However in my default policy setup I have not given the *dpkg_t* domain (which is used by the *dpkg* program when it is run administratively) the privilege of changing the SID of files. So the *setfiles* program needs to have the type *setfiles_exec_t* to trigger an automatic domain transition to the *setfiles_t* domain.

To solve this issue I have the *preinst* script (the script that is run before the package is installed) of the *selinux* package rename the */usr/sbin/setfiles* to */usr/sbin/setfiles.old* on an upgrade. Then the */etc/dpkg/postinst.d/selinux* script will run the old version if it exists.

Here's the relevant section of the *selinux.preinst* file:

```
if [ ! -f /usr/sbin/setfiles.old -a \
    -f /usr/sbin/setfiles ]; then
  mv /usr/sbin/setfiles /usr/sbin/setfiles.old
fi
```

Here's the contents of */etc/dpkg/postinst.d/selinux*. The first parameter to the script is the name of the package that is being installed. Also I have *"grep ..."* included because setfiles currently has some problems with blank lines and */.* which *dpkg* produces.

```
#!/bin/sh

make -s -C /etc/selinux \
  file_contexts/file_contexts

SETFILES=/usr/sbin/setfiles
if [ -x /usr/sbin/setfiles.old ]; then
  SETFILES=/usr/sbin/setfiles.old
fi
dpkg -L $1 | grep ^/.. | $SETFILES -s \
    /etc/selinux/file_contexts/file_contexts
if [ -x /usr/sbin/setfiles.old \
    -a "$1" = "selinux" ]; then
  rm /usr/sbin/setfiles.old
fi
```

4 Running Configuration Scripts in the Correct Context

When a SE Linux system boots the process *init* is started in the domain *init_t*. When it runs the daemon start scripts it uses the scripts */etc/init.d/rc* and */etc/init.d/rcS* on a Debian system (on Red Hat it is */etc/rc.d/rc* and */etc/rc.d/rc.sysinit*). So these scripts are given the type *initrc_exec_t* and there is a rule *domain_auto_trans(init_t, initrc_exec_t, initrc_t)* which causes a transition to the *initrc_t* domain. The security policy for each daemon will have a rule causing a domain transition from the *initrc_t* domain to the daemon domain upon execution of the daemon. This all happens as the *system_u* identity and the *system_r* role.

When the system administrator wants to start a script manually they use the program *run_init* which can only be run from the *sysadm_t* domain, it re-authenticates the administrator (to avoid the possibility of it being called by some malicious code that the administrator accidentally runs) before running the specified script as *system_u:system_r:initrc_t*.

This works fine when the daemon start script is quite simple (most such start scripts just check whether the daemon is already running and then run it with appropriate parameters). However this doesn't work for complex scripts, which may copy files, change sysctl entries via */proc*, and do many other things. An example of this is the *devfsd* package where the start script creates device nodes for device drivers that lack kernel support for *devfs*. Getting this to work correctly required that the code for device node creation be split into a separate file with the same SID as the main daemon (*devfsd_exec_t*) which causes it to run in the same domain as the daemon (*devfsd_t*). Such changes will probably have to be made to about 5% of daemon start scripts.

But that is part of the standard proceedure of correctly setting up SE Linux. The package specific part comes when the scripts have to be started from the package installation. To get the correct domain (*initrc_t*) for the scripts I use the rule *domain_auto_trans(dpkg_t, etc_t, initrc_t)* which causes the *dpkg_t* domain to transition to

the *initrc_t* domain when a script of type *etc_t* is executed. Now the hard part is getting the identity and the role correct when running *dpkg*. For this purpose I have written a customised version of *run_init* to change to the context to *system_u:system_r:dpkg_t*, *system_u:system_r:apt_t*, or *system_u:system_r:dselect_t*, for the programs *dpkg*, *dselect*, and *apt-get* respectively.

The *apt_t* and *dselect_t* domains are only used for selecting and downloading packages, and then executing *dpkg*, which triggers an automatic transition to the *dpkg_t* domain.

5 Managing the Configuration Files

For normal configuration files in Debian (almost every file under /etc and some files in other locations) the file is registered as a *conffile* in the packaging system, and the package status file contains the MD5 checksum of the file. If a file is changed from it's original contents (according to an MD5 check) at the time the package is upgraded and if the new version has a different set of data for the file than that which was provided by the old version of the package (according to MD5) then the user will be asked if they want to replace the old file (with a default of no). However if the new version of the package contains different content and the old content was not changed, then the user will get the new content without even being informed of the fact!

This is OK for many files, but the idea of a file from your audited security configuration being replaced with one you've never seen is not a pleasant one! This is only the first problem with managing policy files, the next problem is the size of the database for the sample policy. If you are using an initial RAM disk (initrd) then you must have the policy database on the initrd. The default initrd size of 4 megabytes is not large enough to accomodate the usual modules and the complete sample policy.

So what we need to solve this is a way of having a set of sample policy files (one per *domain*), of which not all will be used, and when new policy files are added or existing files are changed the user must be prompted as to whether they want to add the new files or apply the changes. Also when adding new policy the matching entries have to be added to the database used by *setfiles* for setting the file context.

In the latest versions of the sample policy the *Makefile* creates a configuration file for *setfiles* to match the program configuration files used. For every application policy file *domains/program/%.te* the matching file *file_contexts/program/%.fc* will be used as part of the configuration. This change will solve the issue of determining the configuration for *setfiles*, but it doesn't entirely solve the problem. One issue with this is that when a file is added to or removed from the configuration the appropriate changes need to be made to the file system. If you make an addition to the policy before installing a new package (the correct proceedure) then you can usually get away without this as long as none of the files or directories previously existed, however this is not always the case, especially when files are diverted or when dealing with standard directories such as */var/spool/mail* which will exist even if you have not installed any software to use them! It should not be that difficult to write a program to relabel the files matching the specifications of the added policy, the question is whether policy additions are common enough to make it worth saving the effort of a relabel. Also there's the risk that a bug in such a program (or it's use) could potentially cause a security hole.

The security policy is comprised of one configuration file per application (or class of application, some domains such as the DHCP client domain *dhcpd_t* are used by multiple programs which perform similar functions). Also sometimes an application requires multiple domains which will therefore be defined in the one file, for example my current policy for Postfix has eleven domains (which is excessive, I plan to reduce it to three or four once I've determined exactly what is required). One problem I faced with this is the issue of what to do when one domain needs to interact with another domain, for example the *pppd* process often needs to run *sendmail -q* to flush the mail queue when it establishes a connection. This requires the policy statement *domain_auto_trans(pppd_t, sendmail_exec_t, sysadm_mail_t)*, previously such a statement would be put in either the *sendmail.te* file or

the *pppd.te* file, thus making one of them depend on the other. This is a bad idea because there's no reason for either of these programs to depend on the other. The solution I devised is based on the M4 macro language (which was already used for simpler macro functionality in producing the policy file). I created a script to define a macro with the name of each application policy file that is used. So the solution to the PPP and Sendmail problem is to put the following in the *pppd.te* file:

```
ifdef(‘sendmail.te’,
‘domain_auto_trans(pppd_t, sendmail_exec_t
                  , sysadm_mail_t)’)
```

The next problem, is how to effectively manage things so that when I ship a new and improved sample policy the administrator can update it without excessive pain.

The current method involves running *diff -ru* and then copying files if you like the changes. This is excessively painful even when managing one or two SE Linux machines! So it obviously won't scale to serious production. I plan to write a Perl script to manage this, the first thing it has to do is track when the administrator doesn't want a policy file. When a file is removed then the fact that the user has chosen not to have that file installed should be recorded, and they should not be prompted to re-install it on the next upgrade. However if the sample policy is upgraded and a new file has been added then they should be asked if they want to install it. Then when a file in the sample policy changes and it is a file that is installed the user should be asked if they want the new file copied over their existing file (and they should be provided with a *diff* to show what the changes would be). Finally if such changes involve the file configuration for *setfiles* then the user should be asked whether they want to relabel the system.

The people who are working on Red Hat packaging are considering other ways of managing the versions of configuration files, one of which involves having symbolic links pointing to the files to be used, if you decide to use your own version instead of one of the supplied policy files then you can change the sym-link.

6 Managing Device Nodes

In Linux there are two methods of managing device nodes. One is the traditional method of having */dev* be a regular directory on the root file system and have device nodes created on it with *mknod*, the other is the *devfs* file system which allows the kernel to automatically create device nodes while the *devfsd* process automatically assigns the correct UID, GID, and permissions to them.

On a traditional (non-devfs) system running SE Linux the device nodes will be labelled in the same way as any other file. On a devfs system things are different, the devfs policy database contains rules for labelling device nodes. However this has some limitations, one being that when the policy database does not have an entry for the device node at the time it is created, then it will never be labelled. Another is that every *type* listed in the devfs configuration rules must be defined, which can cause needless dependencies.

To address these issues I wrote a module for devfsd which adds support for SE Linux. This allows you to change the mapping of SIDs to device nodes and re-apply it at any time, and if a security context listed in the configuration file does not exist in the policy then an error will be logged and the system will continue working.

This is especially useful for the case of an *initrd* as the types for all the possible device nodes won't need to be in the ram disk.

7 Work To Be Done

Initial RAM Disk

When using an initrd to boot a modular kernel the security policy database must be stored on the initrd. The problem is that the default initrd size is 4M, which does not leave much space when libc6 is included, often not enough for the policy you want. Also even if the policy does fit you won't really want to have such a large initrd image. If you are

installing SE Linux on a single PC, or even on a network of similar PCs then you are best advised to build a kernel with all modules needed for booting statically linked and not use an initrd. However this is not possible for a distribution vendor who has to support a huge variety of hardware.

Another problem with using an initrd for storing the policy is that when you generate a new policy you then have to regenerate the initrd to avoid having your changes disappear on the next boot, of course a boot script could easily load the updated policy from the root file system before going to multi-user mode. But it is wasteful to have a large policy on the initrd that you then discard before ever using much of it.

The solution is to have a small policy that contains all the settings needed for either the first stage of boot, or alternately for running recovery tools in case a failure prevents the machine from entering multi-user mode. Then after the machine has passed the first stages of the boot process a complete policy can be loaded from the root file system, as long as the two policies don't conflict in any major way this should work well. NB A Major policy conflict is a situation where the initrd defines domains that aren't defined in the new policy and processes are executed in such a domain.

The latest release of SE Linux supports automatically re-loading the policy when the real root file system is mounted. Now all that needs to be done is for someone to write a mini-policy to install on the initrd.

Polishing run_init

Stephen Smalley has suggested that we develop a *run_init* program that incorporates the functionality of my modified program as well as of the original *run_init* program in a more generic fashion. It is apparent that other people will have similar needs for programs to execute programs under a different domain, role, and maybe identity. It is better that one program do this than to have many people writing programs for such things.

Also currently my program is hard-coded for the names of the Debian administration programs. An improved program should handle the needs of Debian, RPM, and the regular run_init functionality.

Writing Sample Policy Files

Currently any serious system will require policy files that are not in the sample policy. This forces everyone who uses SE Linux to start by writing policy files (which is the most difficult and time consuming task involved with the project). Currently we are writing new sample policy files for the variety of daemons and applications, and developing new macros for writing policy files quickly. With the new macros policy files are on average half the size that they used to be (and I aim to reduce the size again by new macros). The macros allow short policy files which are easy to understand, and therefore the user can easily determine how to make any required changes, or how to write a policy file for a new program based on existing programs.

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

I would like to thank Stephen Smalley for being so helpful when I was learning about SE Linux, and Dr. Brian May for checking my early packages and giving me some good advice when I first started.

Also thanks to Dr. May, Stephen Smalley, and Peter Loscocco for reviewing this paper.

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/`