Writing SE Linux policy HOWTO

Faye Coker
faye@lurking-grue.org
Last update: March 17, 2004

This document continues on from the Getting Started with SE Linux HOWTO, and covers writing SE Linux policy as well as discussing configuration files you will be dealing with. It is aimed at people starting out with writing their own SE Linux policies. If you have not already done so, please read the Getting Started with SE Linux HOWTO in order to become familiar with basic concepts. Any mention of "old SE Linux" refers to the original release of SE Linux for 2.4.x kernels. "New SE Linux" refers to SE Linux for 2.6.x kernels for which a backport is available for 2.4.

Please make sure you also read the NSA's document called Configuring the SE Linux Policy as material in this HOWTO refers to its contents.

This HOWTO tries to be as basic as possible. With learning how to write SE Linux policy, it's mostly a matter of just getting in there and doing it, as many things are not documented at this time. Keep practising, look at existing policies, study the kernel log messages. A lot of what you try might be guesswork which is perfectly okay, because things will gradually fall in to place.

This document has been tested on a test system but more guinea pigs are always welcome. Please email me if you run in to problems when following my instructions.

Table of Contents

---

1. Introduction

In "Getting Started with SE Linux HOWTO", you were introduced to Security Enhanced Linux. This document goes a little further than the "basic stuff" such as installation, adding a new user and terminology. SE Linux contains a number of policy files relating to things you want to do, such as executing certain programs or accessing certain files or directories, and this document will cover writing/editing of those policy files. This document can be read from beginning to end, or used as a reference for various things you might require an explanation for.

1.1. Feedback

Comments on this document are welcome. Please email faye@lurking-grue.org

1.2. Disclaimer

This document is a guide only. I strongly recommend you install SE Linux on a test machine before deploying on a production server. Please read "Getting Started with SE Linux HOWTO" to become familiar with basic concepts.

---

2. All about policies

In this section we'll discuss what policies are, what you can do with them, how they're created and how you make them take effect.

2.1 What is a policy?

Policies are a set of rules governing things such as the roles a user has access to; which roles can enter which domains and which domains can access which types. You can edit your policy files according to how you want your system set up. The purpose of SE Linux is to enforce policies, so policies form the core of SE Linux. The default policy is to deny everything and every operation has to be explicitly permitted in a policy file.

2.2 What can you do with policies?

Polices allow you the flexibility to configure your system as you wish. You may choose to have user A access both the user_r and sysadm_r roles, and have user B access the user_r role only. Policies can be as strict or as lenient as you require. Policies can also control what programs can do, and how programs can interact with each other such as controlling the accessing of files, programs tracing eachother, and sending signals. For example, with regard to accessing files you can have a policy for files created under /tmp so that one domain creates them, but another domain can't access them.

2.3 How are policies created, and how do they take effect?

Policies must be compiled in to binary form before they can be used. In order to do this, you must run `make` in the policy directory, which is `/etc/selinux/` for Debian and `/etc/security/selinux/src/policy` under Red Hat. The process of compiling the SE Linux policy is as follows:

1) *The policy configuration files are concatenated together.*
The policy configuration files end in `.te` and are found in the policy directory and subdirectories under that.

2) *The **m4** macro processor is applied to the result of the above concatenation, which then creates the `policy.conf` file.*
The `policy.conf` file is found in the policy directory, and contains things such as the definition of types, domains, rules for what each domain can do, roles, users, what roles users can access and lots of other stuff.

3) *The **checkpolicy** policy compiler is run on* `policy.conf`.
This results in the creation of the `policy.VERSION` file, where VERSION is the version number. `policy.VERSION` is installed in to the `/etc/security/selinux` directory by running the command `make install` in the policy directory. This policy will then be loaded on the next reboot. If however, you wish to make a runtime change to your policy, you can run the command `make load` in the policy directory so that you can load the new policy in to a running kernel.

2.4 How are decisions made?

When you wish to perform a certain operation on a SE Linux system, your ability to do so is determined by your security context, the class of the object you're trying to access and the type of that object. For instance, a file has a class of file, a directory has a class of dir, and a Unix domain socket has a class of sock_file. The type of the object you're trying to access is predetermined by the policy. Let's say that as unprivileged user faye, I try to do `ls -l /etc/shadow`. Here is what gets logged (check the output of the dmesg command as sysadm_r) when I do this:

```
avc:  denied  { getattr } for  pid=10387 exe=/bin/ls path=/etc/shadow dev=03:03 ino=129766 \
scontext=faye:user_r:user_t tcontext=system_u:object_r:shadow_t tclass=file
```

The Getting Started With SE Linux HOWTO document discussed these kinds of messages so I won't talk about them in depth here, instead I'll just talk about the bits relevant to this section. The source context (scontext) of the user running `ls -l /etc/shadow` is faye:user_r:user_t. So the identity is "faye" and I'm in the user_r role in the user_t domain. The target context (tcontext) is system_u:object_r:shadow_t so in other words, /etc/shadow has the type shadow_t. The class (tclass) of the target is file, so we know that /etc/shadow has a class of file. The entire avc message above shows my attempt to stat, or "getattr" /etc/shadow has failed.

So where is it specified that /etc/shadow has the type shadow_t? The answer is in the file `file_contexts` under the directory file_contexts, which is under your policy source directory. If we grep for /etc/shadow in this file we see

```
/etc/shadow.*           --      system_u:object_r:shadow_t
```

The specification for /etc/shadow's type is also found in the `types.fc` file, which is compiled to produce the file_contexts file. In this example, files called /etc/shadow.* have the system_u identity, as any files "owned" by the system have system_u. object_r is the role assigned to files, which doesn't mean all that much as roles are not relevant to files.

---

3. `policy.conf`, checkpolicy, the Makefile

SE Linux has a number of configuration files which you will find yourself editing at some point. The next few sections will discuss the more commonly used and edited files.

The main config file is `policy.conf` , located in your policy source directory. This file is comprised of the files ending in .te being joined together (see Section 2.3). In the normal course of operation you wouldn't edit policy.conf as it is automatically generated by running "make load", but you could edit it if you wanted to make a quick change such as doing a test.

Files that make up policy.conf will be discussed further on in this document.

3.1 checkpolicy

checkpolicy is the policy compiler, and is run during a "make reload" operation. checkpolicy's main task is to compile the policy, but you can also use it to query the policy. If I run checkpolicy as user_t I see:

```
faye@kaos:/etc/selinux$ checkpolicy
checkpolicy:  loading policy configuration from policy.conf
security:  4 users, 5 roles, 683 types
security:  29 classes, 71806 rules
checkpolicy:  policy configuration loaded
```

This tells me I have four users, and in my `/etc/selinux/users` file they are faye, root, system_u and user_u. I have 5 roles which are user_r, sysadm_r, staff_r, and system_r. Where's the fifth role? Running the command

```
grep ^role policy.conf | cut -f2 "-d "|sort -u
```

shows I only have the four roles mentioned. The fifth role is object_r, which is the role assigned to all files, and is implicitly defined (it doesn't exist in the actual policy). Note that roles for files are not relevant and that if a file requires a level of protection, a specific type will be assigned to that file, such as shadow_t for /etc/shadow.

The above example also shows I have 683 types (domains and types for files, directories and so forth), 29 classes (such as file, dir, unix_stream_socket and so on) and 71806 rules.

3.2 the Makefile

The Makefile in the policy source directory provides for the following operations to be performed:

install
Running "make install" compiles and installs the policy, but does not load it. You would run this if you were not running an SE Linux kernel and wanted to install the policy so that when you next boot in to an SE Linux kernel, the policy would loaded.

load
Running "make load" compiles, installs and then loads the policy configuration. You don't need to reboot your machine.

reload
Running "make reload" compiles, installs and loads or reloads the policy configuration. When the Makefile loads a policy, a flag file called "load" is created in the tmp directory under your policy source directory. A "make load" won't do anything if this flag file exists AND you haven't changed anything more recently than the flag file's creation time, but you can reload.

relabel
Running "make relabel" relabels filesystems based on the file contexts configuration. The file contexts configuration file is located in the file_contexts directory under your policy source directory.

policy
Running "make policy" compiles the policy locally for testing/development. This results in the policy being compiled but not actually installed.

4. Attributes: the `attrib.te` file

This section will discuss attributes, which are a way of grouping sets of types. The `attrib.te` file will be briefly examined. This file is located in your policy source directory, and contains attribute declarations for domains and types. Editing this file is not very common, however if you wanted to add a new attribute, you would edit it. The comments at the top of attrib.te state "a type attribute can be used to identify a set of types with a similar property. Each type can have any number of attributes, and each attribute can be associated with any number of types." Domains attributes are groupings of domains, just as attributes are groupings of types.

Examples:
The *domain* attribute identifies every type that can be assigned to a process. This attribute refers to all processes that could be run, such as ps, top, inetd and so forth.

The *privuser* attribute identifies every domain that can change its SELinux user identity. Note that we are talking about SE Linux user identity and not the standard Unix uid. Running the command `grep ^type.*privuser policy.conf` shows that the domains which can change their identity include sysadm_su_t, initrc_su_t, staff_su_t, run_init_t, local_login_t, remote_login_t, sshd_t, sshd_extern_t and xdm_t.

The *privrole* attribute identifies every domain that can change its SELinux role. A domain can spawn processes that have a different role. Take **newrole** for example. The point of newrole is to change to another role, so the privrole attribute needs to be assigned to newrole_t in order for this to happen. privrole only allows changing to other user roles. priv_system_role allows you to change to system_r.

The *privowner* attribute identifies every domain that can assign a different SELinux user identity to a file, or that can create a file with an identity that's not the same as the process identity. Using passwd_t as an example, the passwd_t process has the identity of the user running it, and it wants to relabel /etc/shadow with the system_u identity, thereby requiring privowner.

The *userpty_type* attribute identifies all non-administrative devpts types such as user_devpts_t and staff_devpts_t . For instance, if I run the command `ls --context /dev/pts` on my system, I will see

```
crw-------  faye     staff    faye:object_r:staff_devpts_t     0
[snip]
```

Here, /dev/pts/0 has the type staff_devpts_t .

The *sysadmfile* attribute identifies all types assigned to files that should be completely accessible to administrators. Note that shadow_t is not by default accessible to administrators. It can be accessed by something like, for instance, setfiles, depending on what you're trying to do.

The *fs_type* attribute identifies all types assigned to filesystems (not limited to persistent filesystems). security_t applies to the /selinux filesystem in the new SE Linux.

The *ptyfile* attribute identifies all types assigned to ptys. The explanation for the ttyfile attribute also applies here, but with pty's. Running `ls --context 'tty'` in an xterm would show the type of the pty device you are attached to, for e.g.

```
faye@kaos:/etc/selinux$ ls --context 'tty'
crw-------  faye     faye     faye:object_r:user_devpts_t      /dev/pts/1
```

If I then switch to sysadm_r and run the same command, I see

```
faye@kaos:/etc/selinux$ newrole -r sysadm_r
Authenticating faye.
Password:
faye@kaos:/etc/selinux$ id
uid=1000(faye) gid=1000(faye) groups=1000(faye),20(dialout),25(floppy),29(audio),30(dip) \
context=faye:sysadm_r:sysadm_t
faye@kaos:/etc/selinux$ ls --context 'tty'
crw-------  faye     faye     faye:object_r:sysadm_devpts_t    /dev/pts/1
```

Note that my pty is now labelled with the type sysadm_devpts_t.

The *login_contexts* attribute identifies the files used to define default contexts for login types (e.g., login, cron). Default context for login types are contained in the file `/etc/security/default-contexts`.

5. User related files

This section will discuss policy files related to actual users on the system, and where you can determine their level of access. In "Getting Started with SE Linux HOWTO" we saw that "an identity under SE Linux is not the same as the traditional Unix uid (user id). They can coexist together on the same system, but are quite different. identities under SE Linux form part of a security context which will affect what domains can be entered, i.e. what essentially can be done. An SE Linux identity and a standard Unix login name may have the same textual representation (and in most cases they do), however it is important to understand that they are two different things."

5.1 The `users` file

The `users` file, found in your policy source directory for your distribution, such as `/etc/selinux/users` under Debian, contains definitions for each user that is to be recognised by your SE Linux system. If a user identity is explicitly named in this file, their user identity will form the first part of their security context. A security context is made up of the identity, role and domain or type. You can check your own current security context by running the **id** command under SE Linux. If a user identity is not named in the `users` file, they will be assigned the **user_u** identity.

Following are some example entries in a `users` file.

```
user root roles { staff_r sysadm_r };
```

This entry defines user identity root, and allows root to enter the staff_r and sysadm_r roles. The **newrole** command can be used to change user roles, or root (in this example) can choose to enter one of the roles above when logging in at the console.

```
user faye roles { staff_r sysadm_r };
```

Again, this entry defines the user identity faye. As with root, faye is also able to enter the sysadm_r role, which is the system administrator role. These two examples show you that the traditional root user does not necessarily have to have sysadm_r privileges just because they are user root, and that another user (such as faye) can have access to the system administrator role. If the roles definition for user root did not have sysadm_r, faye would be more powerful than root.

```
user foo roles { user_r };
```

User foo is defined, and they only have access to the user_r role, which is the general unprivileged user role.

```
user system_u roles system_r;
```

The system_u identity is the user identity for processes and objects such as files, directories, sockets and so on. You should not assign the system_u identity to a user process, as system_u is for daemons. If a user has system_u, they may potentially have access to the system_r role and any daemon domains.

5.2 The `user.te` file

This file is found in the subdirectory "domains" under your policy source directory, such as `/etc/selinux/domains` on Debian. `user.te` contains the unprivileged domains for the users on your SE Linux system. In this file you will see the line

```
full_user_role(user)
```

This line allows all access necessary for the user role to do standard things such as run bin_t programs in their home directory, assign user_home_dir_t to the user's home directory and user_home_t to directories under that. The lines

```
full_user_role(staff)
allow staff_t unpriv_userdomain:process signal_perms;
can_ps(staff_t, unpriv_userdomain)
allow staff_t { ttyfile ptyfile tty_device_t }:chr_file getattr;
```

defines the domain staff_t. The second line allows the staff_t domain to send signals to processes running in unprivileged domains such as user_t and staff_t. The third line allows staff_t to run ps and see processes in the unprivileged user domains. staff_t is able to run ps and see everything in user_t and other user domains if any, whereas user_t can not. The fourth line allows staff_t to access the attributes of any terminal device.

```
dontaudit unpriv_userdomain sysadm_home_dir_t:dir { getattr search };
```

This line says not to audit unprivileged user domain (such as user_t) attempts to acces the /root directory by doing something like a ls -l or cd /root command, or trying to access a file under the /root directory. The line can also be

read as "dontaudit source destination:destination class {what was attempted }".

```
# change from role $1_r to $2_r and relabel tty appropriately
define('role_tty_type_change', '
allow $1_r $2_r;
type_change $2_t $1_devpts_t:chr_file $2_devpts_t;
type_change $2_t $1_tty_device_t:chr_file $2_tty_device_t;
')
```

The first (uncommented) line defines the macro role_tty_type_change. This macro allows you to change roles and then relabel the tty you're using (such as when you use the newrole command to get from staff_r to sysadm_r and your tty changes). The second line says that $1_r is allowed to transition to $2_r (so $1_r might be staff_r and $2_r is sysadm_r). The relabelling of the tty is handled by the third and fourth lines.

```
ifdef('newrole.te', '
#
# Allow the user roles to transition
# into each other.
role_tty_type_change(sysadm, user)
role_tty_type_change(staff, sysadm)
role_tty_type_change(sysadm, staff)
')
```

This block says that if newrole.te is defined (i.e. if the file exists) then allow the roles shown to transition to eachother. These statements are the invocations of the role_tty_type_change macro as defined previously.

5.3 The `user_macros.te` file

This file contains macros for user login domains which are domains users get upon logging in, as opposed to the domain of something like a cron job. Other non-login domains include those for gpg and irc, among others. I am using extracts of my own user_macros.te file as the example here (not the whole file). This file is located under the macros subdirectory of your policy source. I will break this file up in to little chunks for ease of explanation. Before I do that, I'd like to provide a brief definition of "capabilities": the ability to perform certain privileged operations.

5.3.1 *Macros for user login domains*

```
 # user_domain() is also called by the admin_domain() macro
undefine('user_domain')
define('user_domain','
```

The above extract defines the macro `user_domain`

```
# Use capabilities
allow $1_t self:capability { setgid chown fowner };
dontaudit $1_t self:capability { sys_nice fsetid };
```

"Capabilities" in this snippet means the capability to change the group id (setgid). A program can call setgid() to change its group id, but only if it has the capability to do so in the first place, as well as not having given up the capability previously. A program can't call setgid() if this capability isn't granted. An in-depth discussion of capabilities is beyond the scope of this document, however please see `/usr/include/linux/capability.h` for more information. The first line allows the $1_t type the capability to setgid. $1 is the first parameter passed from the calling code (the code that called the macros). The second line says not to audit sys_nice (the ability to increase scheduling priority) or fsetid (related to controlling setuid and setgid files). These are capabilities requested repeatedly, so this line says not to audit the fact they're being requested.

```
# Type for home directory.
ifelse($1, sysadm, '
type $1_home_dir_t, file_type, sysadmfile, home_dir_type, home_type;
type $1_home_t, file_type, sysadmfile, home_type;
tmp_domain($1)
', '
type $1_home_dir_t, file_type, sysadmfile, home_dir_type, user_home_dir_type, home_
type, user_home_type;
type $1_home_t, file_type, sysadmfile, home_type, user_home_type;
```

The above says that if $1 is sysadm then do the first block, otherwise do the second block, where both define types for

home directories and tmp_domain (the macro for /tmp file access). So, the first block relates to sysadm stuff, and the second relates to the rest.

```
 # do not allow privhome access to sysadm_home_dir_t
file_type_auto_trans(privhome, $1_home_dir_t, $1_home_t)
tmp_domain($1, ', user_tmpfile')
')
```

Here, we don't want privhome domains to access sysadm_home_dir_t. Take procmail for example. When procmail delivers mail it creates a file in a user's home directory. We don't want that happening to a sysadm directory (/root). "file_type_auto_trans" is the way to set the default type for a new file, otherwise it will be the same type as the directory it is created in. Normally when you create a file, it will have the same type as the directory it is created in. There are two ways to have a different type to that of the directory at creation time. The first is through open_secure() and the second is through file_type_auto_trans. For the first to be allowed, you need to have a file_type_trans rule that permits it, and then file_type_auto_trans makes it the default.

```
# allow ptrace
can_ptrace($1_t, $1_t)
```

This allows the first parameter ($1_t) to ptrace, or trace the process of the second parameter (also $1_t). This line therefore allows $1_t to ptrace itself.

```
# Create, access, and remove files in home directory.
file_type_auto_trans($1_t, $1_home_dir_t, $1_home_t)
allow $1_t $1_home_t:dir_file_class_set { relabelfrom relabelto };
```

Here, a process in domain $1_t creates a file under a directory of type $1_home_dir_t and by default the created file is of type $1_home_t. The second line allows $1_t to relabel a file of type $1_home_t to something else, and to change the type from something else to $1_home_t.

```
# Bind to a Unix domain socket in /tmp.
allow $1_t $1_tmp_t:unix_stream_socket name_bind;
```

This allows $1_t to bind to a Unix domain socket, so it can then receive connections from other processes. $1_tmp_t is the type of the tmp_domain. name_bind allows $1_t to bind to a name under /tmp.

5.3.2 *Macros for ordinary user domains*

Everything below refers to user_t.

```
undefine('full_user_role')
define('full_user_role', '

# user_t/$1_t is an unprivileged users domain.
type $1_t, domain, userdomain, unpriv_userdomain, web_client_domain;

# $1_r is authorized for $1_t for the initial login domain.
role $1_r types $1_t;
allow system_r $1_r;

# Grant permissions within the domain.
general_domain_access($1_t);
```

Define the macro full_user_role. Define the type $1_t/user_t and give it the four attributes listed. $1_r/user_r is able to have $1_t/user_t. system_r is then allowed access to $1_r/user_r. general_domain_access allows $1_t to see process in $1_t, see files in /proc/# among others (check the file `core_macros.te`).

```
# Read /etc.
allow $1_t etc_t:dir r_dir_perms;
allow $1_t etc_t:notdevfile_class_set r_file_perms;
allow $1_t etc_runtime_t:{ file lnk_file } r_file_perms;
```

Allow user_t to read etc_t (the type of /etc). You can see and read files under /etc and do things such as ls -l commands on /etc. If you look at the file `core_macros.te` you will see that notdevfile_class_set related to non-device file classes such as files, symlinks, socket files and fifo files. etc_runtime_t is the type for certain files in /etc (grep for "etc_runtime_t" in the file file_contexts).

```
undefine(‘in_user_role’)
define(‘in_user_role’, ‘
role user_r types $1;
role staff_r types $1;
’)
```

Define the macro in_user_role. A domain can be used in any of the user roles. The macro must be called in the .te file of the domain concerned. Look at the file passwd.te (for the passwd program). The in_user_role macro is invoked and has the passwd_t parameter passed to it. Going back to the above snippet, we can now say

```
role user_r types passwd_t;
role staff_r types passwd_t;
```

meaning that user_r and staff_r can run the passwd program. Remember that if you add a new role, you must edit the in_user_role macro here.

---

6. System administrator related files

This section will discuss the policies related to the sysadm_r role, i.e., the system administrator. We have already seen how an SE Linux identity can be granted sysadm_r in section 4.1.

6.1 The `admin_macros.te` file

The `admin_macros.te` file contains macros for the system administration domains.

```
undefine(‘admin_domain’)
define(‘admin_domain’,‘
# Inherit rules for ordinary users.
user_domain($1)
```

Define the macro admin_domain and allow it to have the same rules as user_t. $1 in this case would be sysadm.

```
allow $1_t policy_config_t:dir { getattr search };
allow $1_t policy_config_t:file getattr;
```

Allow sysadm_t to getattr (things such as *ls -l*) and search files and directories under a directory that has a type of policy_config_t.

```
allow $1_t kernel_t:system syslog_read;
```

Allow sysadm_t to read the system logs. kernel_t is the type for the kernel itself. system is the class of the operation, the operation being to read the syslog.

```
# Use capabilities other than sys_module.
allow $1_t self:capability ~sys_module;
```

Allow sysadm_t to use all capabilities apart from sys_module, which is used to load modules.

```
# Get security policy decisions.
can_getsecurity($1_t)
```

If you look at the file core_macros.te (under the macros directory) and search for can_getsecurity, this is what you see:

```
# can_getsecurity(domain)
#
# Authorize a domain to get security policy decisions.
#
define('can_getsecurity','
allow $1 security_t:dir { read search getattr };
allow $1 security_t:file { getattr read write };
allow $1 security_t:security { check_context compute_av compute_create compute_relabel compute_user };
')
```

Here, $1 is allowed to read, search and get attributes of a directory of type security_t (your policy source directory). $1 can also get attributes, read and write files in a directory of type security_t. Finally, $1 cancheck context validity, check whether the policy permits the source context to access the target context, compute a context for the labelling of a new object, compute the new context when relabelling an object, and to determine which user contexts can be reached from a given source context.

```
# Change system parameters.
can_sysctl($1_t)
```

sysadm_t is able to modify sysctl parameters, which is basically everything under `/proc/sys`. If you run the command `grep ^type.*sysctl_type policy.conf` you'll see the types that have the attribute sysctl_type.

---

7. the file_contexts file

The `file_contexts` file contains security contexts which are applied to files on the system when a security policy installed. This file is read by the *setfiles* program and uses the information to label files. Below are some examples and explanations.

```
# The security context for all files not otherwise specified.
/.*                             system_u:object_r:file_t
```

This line sets the security context on files that do not have a specified context. system_u is the idenity for system processes and daemons and is the default identity for files owned by the system.

```
# The root directory.
/                       -d      system_u:object_r:root_t
```

Set the context with a type of root_t for the actual root directory (specified by the -d). `/mnt` and `/initrd` also have the type root_t.

```
/home/[^/]+             -d      system_u:object_r:user_home_dir_t
/home/[^/]+/.+                  system_u:object_r:user_home_t
```

For the actual `/home` directory, set the type to user_home_dir_t. For files underneath it, set the type to user_home_t.

You should be able to get a general understanding of everything else in this file, and it does help to have a good understanding of regular expressions.

In the middle column, you may see -- which refers to a regular file. -d refers to a directory. Nothing listed means anything is matched. If you do an "ls -l" command, the first character of the first column of output is what appears in the middle column. So if something was a symbolic link you'd see -l, -b for a block device and so forth.

---

8. the types directory

The `types` directory contains definitions of types, broken up in to the following files:

8.1 device.te

This file contains the types for device nodes.

```
type device_t, file_type;
```

This line defines the type device_t for `/dev`. file_type is the attribute that is used for all types for files and directories. If you search for /dev in the file `file_context` you will see its type is set to device_t.

```
 type null_device_t, file_type, device_type, mlstrustedobject;
```

Defines the type null_device_t for `/dev/null`. The device_type attribute identifies all types assigned to device nodes. mlstrustedobject is not used at this time.

8.2 devpts.te

This file contains the types for pseudo ttys.

```
type devpts_t, fs_type, root_dir_type;
```

Set the type of the devpts filesystem (devpts_t) and the type of the root directory of that filesystem.

8.3 file.te

This file contains the types for files.

```
type unlabeled_t, sysadmfile;
```

Unlabeled objects have the type unlabeled_t. Any time you change the policy to remove the definition of a type, everything that uses that type becomes unlabeled.

8.4 network.te

This file contains the types for networking.

```
type netif_t, netif_type;
type netif_eth0_t, netif_type;
type netif_eth1_t, netif_type;
type netif_eth2_t, netif_type;
type netif_lo_t, netif_type;
type netif_ippp0_t, netif_type;
```

The netif types are used for network interfaces.

8.5 nfs.te

This file contains types for NFS usage.

```
type nfs_t, fs_type, root_dir_type;
```

nfs_t is the default type for NFS file systems and their files. Set the root directory of the NFS file system to be of type nfs_t.

8.6 procfs.te

This file contains types for the proc file system.

```
type proc_t, fs_type, root_dir_type;
type proc_kmsg_t;
type proc_kcore_t;
```

proc_t is the type of the proc file system. proc_kmsg_t is the type for `/proc/kmsg`. proc_kcore_t is the type for `/proc/kcore`.

8.7 security.te

This file contains types for security stuff for SE Linux.

```
type security_t, fs_type;
type policy_config_t, file_type;
type policy_src_t, file_type;
```

security_t is the target type when checking the permissions in the security class. policy_config_t is the type of `/etc/security/selinux/*` and policy_src_t is the type of `/etc/selinux/*` (on Debian).

---

9. the macros directory

We've already covered the files user_macros.te and admin_macros.te. The two other files in the macros directory are core_macros.te and global_macros.te.

9.1 core_macros.te

The core_macros.te file contains macros that are not changed very often, and it is recommended that you don't change them :) This is because core related policy should be the same if you want to share policy. Changing something in this file might render your policy incompatible with what everyone else is doing. Of course, if you don't care about anyone else, go ahead and change it. If you've changed your core_macros.te file, this may result in you having a system that works differently from everyone else, which may not be in your best interests.

Some of the macros contained in this file are macros for groupings of classes and permissions:

```
define('dir_file_class_set', '{ dir file lnk_file sock_file fifo_file chr_file blk_file }')
```

This line defines the macro dir_file_class_set which contains the classes dir (for directories), file (for files), lnk_file (for symbolic links), sock_file (for Unix domain sockets), fifo_file (for named pipes), chr_file (for character block devices) and blk_file (for block devices).

```
define('rw_file_perms', '{ ioctl read getattr lock write append }')
```

This line defines the macros rw_file_perms which contains the permissions ioctl (for ioctl's), read (read file), getattr (get attributes) and then lock, write and append.

9.2 global_macros.te

The global_macros.te file contains macros that are system wide, meaning they are not tied to particular policy files. You can edit this file if you want to, but it probably won't happen often.

```
define('can_setexec','
allow $1 self:process setexec;
allow $1 proc_t:dir search;
allow $1 proc_t:{ file lnk_file } read;
allow $1 self:dir search;
allow $1 self:file { read write };
')
```

Defines the macro can_setexec. $1 is able to set the execute context, so it can set the context of a child process. $1 can search `/proc` and can read files and symlinks in that directory.

9.3 the `macros/program` directory

The `program` subdirectory contains additional macros for programs that need per-user role policy. Programs such as ssh require a per-user role policy as the derived domains are based on the calling user domain. If you look at the file `ssh_macros.te` you'll see

```
define('ssh_domain',‘
# Derived domain based on the calling user domain and the program.
type $1_ssh_t, domain, privlog;
```

So if user_t was the calling user domain, the derived domain will be user_ssh_t. Likewise, if staff_t was the calling user domain, staff_ssh_t will be the derived domain. The line

```
domain_auto_trans($1_t, ssh_exec_t, $1_ssh_t)
```

allows the transition from the calling domain to the derived domain.

---

10. the `flask` directory

The `flask` directory contains the following files:

access_vectors
This file defines the actions that can be performed for various classes. For the file class, you may perform actions such as read, write, link and so forth. For the socket class, you can perform actions like bind (for binding to a socket such as a TCP or UDP socket), listen (for incoming connections), connect and so on. Take a look through this file to familiarise yourself with the different actions various classes may perform.

initial_sids
This file defines the initial SIDS (Security Identifiers). In the old SE Linux, SIDS were used in the userspace interface to the kernel. PSIDs (Persistent SIDs) were used in the kernel code for mapping files to contexts for files and directories on disk. See the NSA's document "Configuring the SELinux Policy" document for more information. In the new SE Linux, the extended attributes contain the context so SIDs and PSIDs are no longer necessary. Even though the new SE Linux uses extended attributes, some initial contexts still need to be defined when a system is started. The initial_sids file contains the initial SID contstants. The file `initial_sid_contexts` in your policy source directory maps these initial SIDS to contexts, and some examples follow:

```
sid kernel      system_u:system_r:kernel_t
sid security    system_u:object_r:security_t
```

The first line defines the initial SID of kernel, and gets the context of system_u:system_r:kernel_t . kernel_t is the type for general kernel code. The second line gives the sid security the context of system_u:object_r:security_t where security_t is the type for the /selinux file system.

security_classes
This file defines the security object classes. These are the classes for things such as files and networking.

An in-depth discussion of the Flask architecture is way beyond the scope of this document, but more information can be found in the NSA document "Configuring the SELinux Policy", particularly the section "Architectural Concepts and Definitions", at http://www.nsa.gov/selinux/doc/policy2/x34.html

---

11. Editing the policy

We'll now get in to the fun part of actually editing SE Linux policy. This takes a fair bit of practise. The best thing to do is just play around with it all. It can be quite a challenge to get started with as a lot of stuff isn't documented and you'll most likely take the trial-and-error approach. Make sure you look at other policies already written in `/etc/selinux/domains/program/` and the corresponding `file_contexts` file in /etc/selinux/file_contexts/program/ .

Here are some tips to keep in mind when you edit policy.

Work out what you want to change/edit.
Try to get a good understanding of what you're going to change. Is there some kind of action you want to allow, such as (for instance) allowing users in the user_r role to be able to see a certain directory not allowed in a default installation? What domain/s would be involved? What macros would you need to call? Look at existing rules to get an idea of syntax. Look up macros in the macros directory to get a feel for what they do.

Create a file where you have custom rules.
I have a file called custom.te in the subdirectory domains/misc/ (under my policy source directory). In this file I

include my own rules which are customised for what I want to do. Call this file something unique so that it won't get overwritten during an upgrade of the policy. custom.te should be fine. You can use this file to test stuff.

Study the kernel messages.
If something isn't happening as you want it to, check out the kernel messages. A big part of writing policy is to study the logs then add or change rules to eliminate errors listed in the logs. If staff_t tries to run tcpdump and the operation is denied for instance, check the logs. You might see something like

```
avc:  denied  { create } for  pid=17824 exe=/usr/bin/traceroute.lbl scontext=faye:staff_r:staff_t tcontext=
:staff_r:staff_t tclass=rawip_socket
```

Here we can see that someone in the staff_t domain tried to execute the traceroute command and was denied. From this log message we can work out that traceroute was running in the domain staff_t but we want it to run in a privileged domain that is able to traceroute. What we would need to do is edit our policy to allow staff_t to transition to traceroute_t (the domain for the traceroute process) when it executes the type traceroute_exec_t (which is the type of the actual traceroute executable, as opposed to the traceroute process that runs after you've executed the traceroute command).

Comment the changes you make and why you made them.
Self explanatory :)

---

12. Basic policy editing examples

Following are some examples of things you can try to get a feel for editing policy. **Don't forget to run `make load` in your policy source directory after saving the policy file.**

12.1 Allow user_t to use tcpdump

If you haven't done so already, create a custom file (let's call it custom.te) in the subdirectory `domains/misc/` under your policy source directory. Add the following lines:

```
# your own comment here
domain_auto_trans(userdomain, netutils_exec_t, netutils_t)
in_user_role(netutils_t)
allow netutils_t user:chr_file rw_file_perms;
```

First of all, we need to be able to transition from the user domain (userdomain here refers to all possible user domains, so user_t, staff_t, sysadm_t and whatever other user domains you may have) to the domain for the actual tcpdump process, which is netutils_exec_t. What the first line is saying is "when the user domain runs the tcpdump executable, an automatic transition is made to the tcpdump process domain which is netutils_t".

The in_user_role macro (defined in the file `user_macros.te`) permits the domain passed as a parameter (in this case netutils_t) to be in all the user roles (such as user_r and staff_r. sysadm_r is an administrative role, not a user role). This line is needed so that any combination of role user_r and domain netutils_t is valid in a security context.

The third line allows the domain netutils_t to access user pty types. netutils_t needs this access so that you can read from and write to your terminal device. chr_file is needed to write to the terminal device.

Exercise 1: Play around with these lines. Comment out the allow line, reload the policy and try to tcpdump. Check the logs to give you hints on why nothing seems to be happening.

Exercise 2: With the allow line commented out, switch to a virtual console (assuming you were using an xterm before) and try to tcpdump from there. If you have not specifically allowed tcpdump access from a tty device, try to make it happen. You should be able to tcpdump from a pty device, but not from a tty device so make the necessary changes and try again.

12.2 Allow user_t to read the `/etc/selinux/` directory.

Ordinarily you probably don't want unprivileged users to see what's in `/etc/selinux/` but for the sake of learning (assuming you're on a test machine playing around with SE Linux) I'll use this example. Edit your custom.te file to

include the following:

```
# your comment here
r_dir_file(user_t,policy_src_t)
```

The r_dir_file allows you to read a directory and files underneath it. user_t is the domain, and policy_src_t is the type of `/etc/selinux` .

Exercise 1: try to access `/etc/selinux` before and after editing custom.te (and reloading the policy). Check the logs to see what's happening.

Exercise 2: from user_t try to access `/boot`. Got "permission denied"? Create a rule that allows user_t to read this directory.

12.3 Creating a new type

In this example we'll create a new file type for ourselves in custom.te so add the following lines and then run the make load command:

```
type ourtype_t,file_type,sysadmfile;
allow staff_t ourtype_t:file { create_file_perms relabelfrom relabelto };
```

We define our new type called "ourtype_t", assign it the attribute file_type and sysadmfile so that the administrator can access it. The second line says that staff_t has full access to files of type ourtype_t (read, write and so forth). The relabelfrom and relabelto mean that staff_t can relabel files of type ourtype_t from and to another type.

Now, in a staff_t role (I'm not going to tell you how to do that, read the Getting Started with SE Linux HOWTO) create a file. Check the security context of that file:

```
faye@kaos:~$ ls -Z foo
-rw-r--r--  faye     faye     faye:object_r:staff_home_t       foo
```

So we see file "foo" has the type staff_home_t. Now change that type to ourtype_t:

```
faye@kaos:~$ chcon -t ourtype_t foo
faye@kaos:~$ ls -Z foo
-rw-r--r--  faye     faye     faye:object_r:ourtype_t          foo
```

Now let's remove the type we just created. Once again, edit your custom.te file to comment out what you just added. Run make load again, and try to look at the file's attributes:

```
faye@kaos:~$ ls -Z foo
ls: foo: Permission denied
```

As sysadm_r run the same ls command:

```
-rw-r--r--  faye     faye     faye:object_r:ourtype_t          /home/faye/foo
```

Now check your logs for the error that was logged when, as staff_r, you tried to access file foo:

```
avc:  denied  { getattr } for  pid=29494 exe=/bin/ls path=/home/faye/foo dev=md7 ino=145445 scontext=faye:s
_r:staff_t tcontext=system_u:object_r:unlabeled_t tclass=file
```

Note that the target context contains the type unlabeled_t for file foo. When we removed type ourtype_t from the policy, any files we created with that type are relabelled to type unlabeled_t. Note that even though ls -Z says the type is ourtype_t the kernel regards it as unlabeled_t as ourtype_t does not exist in the policy.

---

13. Case study: the policy for INN

I will use the policy written for INN (InterNetNews server) to help you get a better idea of writing policy for an application. After you edit and save the policy file, run "make load" in your policy source directory to apply the changes.

There are three files involved with the policy for INN. One is `/etc/selinux/domains/program/innd.te` . The second is the corresponding file_contexts file, `/etc/selinux/file_contexts/program/innd.fc`. The third is the `net_contexts` file.

13.1 The innd.te file

Here is the entire policy. When you start writing policy, you can try and start the daemon. You'll get lots of messages in your logs, so you'll have to create rules to begin eliminating any "denied" messages. These give you clues as to what to do next.

```
#DESC INN - InterNetNews server
#
# Author:  Faye Coker <faye@lurking-grue.org>
# X-Debian-Packages: inn
#
################################

# Types for the server port and news spool.
#
type innd_port_t, port_type;
type news_spool_t, file_type, sysadmfile;


# need privmail attribute so innd can access system_mail_t
daemon_domain(innd, ', privmail')

# allow innd to create files and directories of type news_spool_t
create_dir_file(innd_t, news_spool_t)

# allow user domains to read files and directories these types
r_dir_file(userdomain, { news_spool_t innd_var_lib_t innd_etc_t })

can_exec(initrc_t, innd_etc_t)
can_exec(innd_t, { innd_exec_t bin_t })
ifdef('hostname.te', '
can_exec(innd_t, hostname_exec_t)
')

allow innd_t var_spool_t:dir { getattr search };

can_network(innd_t)

can_unix_send( { innd_t sysadm_t }, { innd_t sysadm_t } )
allow innd_t self:unix_dgram_socket create_socket_perms;
allow innd_t self:unix_stream_socket create_stream_socket_perms;
can_unix_connect(innd_t, self)

allow innd_t self:fifo_file rw_file_perms;
allow innd_t innd_port_t:tcp_socket name_bind;

allow innd_t self:capability { dac_override kill setgid setuid net_bind_service };
allow innd_t self:process setsched;

allow innd_t { bin_t sbin_t }:dir search;
allow innd_t usr_t:lnk_file read;
allow innd_t usr_t:file { getattr read ioctl };
allow innd_t lib_t:file ioctl;
allow innd_t { etc_t resolv_conf_t }:file { getattr read };
allow innd_t { proc_t etc_runtime_t }:file { getattr read };
allow innd_t urandom_device_t:chr_file read;

allow innd_t innd_var_run_t:sock_file create_file_perms;

# allow innd to read directories of type innd_etc_t (/etc/news/(/.*)? and symbolic links with that type
etcdir_domain(innd)

# allow innd to create files under /var/log of type innd_log_t and have a directory for its own files that
# it can write to
logdir_domain(innd)

# allow innd read-write directory permissions to /var/lib/news.
var_lib_domain(innd)

ifdef('crond.te', '
system_crond_entry(innd_exec_t, innd_t)
')
```

The first step is to create the file. Because innd is a daemon listening on a port, we will need to create the type *innd_port_t* and assign it the port_type attribute (remember the attrib.te file?). We also know that a news spool is required, and we want to have a label assigned to the news spool files specific to INN. We call that type *news_spool_t* and give it the attributes of file_type and sysadmfile. sysadmfile is needed because we must grant access to the administrator domain to access those files with type news_spool_t. So from all this, we get

```
type innd_port_t, port_type;
type news_spool_t, file_type, sysadmfile;
```

Next, we need the macro *daemon_domain()* so that we can establish innd as a daemon with its own domain (which is innd_t). We need the attribute *privmail* because innd needs to be able to transition to the system_mail_t domain in order to send mail. Now we have

```
daemon_domain(innd, ‘, privmail’)
```

Above we created the type *news_spool_t* and now we want innd to be able to create the files and directories with that type. The create_dir_file() macro is needed for this so we have

```
create_dir_file(innd_t, news_spool_t)
```

which says that innd_t can create directories and files of type news_spool_t.

We need our unprivileged users to be able to read the news spool files. We also need them to be able to read /var/lib/news/ which has been assigned the type *innd_var_lib_t* (see the innd.fc file), and we need them to be able to read /etc/news/ which has the type *innd_etc_t*. When these rules aren’t plugged in, you’ll get "avc denied" messages in your logs, so from there we work out what labels and rules are needed. Now we have

```
r_dir_file(userdomain, { news_spool_t innd_var_lib_t innd_etc_t })
```

When starting up innd at this point, I received avc denied messages for initrc_d not being able to access innd_etc_t, and innd_t not being to access *innd_exec_t* and *bin_t*. We need the *can_exec()* macro here so that innd_t can execute programs of those types, giving us

```
can_exec(initrc_t, innd_etc_t)
can_exec(innd_t, { innd_exec_t bin_t })
```

An avc denied message in the logs was showing that innd_t could not access *hostname_exec_t*. We could put in a rule allowing innd_t to execute files of type hostname_exec_t but instead we take the following approach:

```
ifdef(‘hostname.te’, ‘
can_exec(innd_t, hostname_exec_t)
’)
```

The ifdef line is used with the *can_exec* macro because if you don’t name hostname.te in the policy, then there will be no type *hostname_exec_t* and the policy won’t compile. We then use the *can_exec()* macro as before.

We now want innd_t to be able to search and get the attributes of directories with the type *var_spool_t* and looking for *var_spool_t* which is the /var/spool directory.

```
allow innd_t var_spool_t:dir { getattr search };
```

We know that innd_t will need network functionality so we need the following:

```
can_network(innd_t)
```

After that, we’ll place all the network related stuff together to make for easier reading.

```
can_unix_send( { innd_t sysadm_t }, { innd_t sysadm_t } )
```

The *can_unix_send()* macro is used for sending Unix datagrams. In the line above we are saying "innd_t can send to and receive from itself and the sysadm_t domain, and sysadm_t can send to and receive from innd_t and itself".

We now need innd_t to be able to create socket permissions in itself (that is, in its own domain of innd_t) so we add

```
allow innd_t self:unix_dgram_socket create_socket_perms;
allow innd_t self:unix_stream_socket create_stream_socket_perms;
```

innd_t will also need to be able to establish a stream connection to itself and for this we need the following macro and parameters:

```
can_unix_connect(innd_t, self)
```

innd_t will need read-write file permissions on anonymous pipes (created by the pipe() system call) in the innd_t domain. For this we include

```
allow innd_t self:fifo_file rw_file_perms;
```

innd_t will need to bind to a tcp_socket so we need

```
allow innd_t innd_port_t:tcp_socket name_bind;
```

innd_t will need the following capabilities listed. See /usr/include/linux/capability.h for more information on what each capability does. innd_t will also need to be able to change the nice levels of innd processes.

```
allow innd_t self:capability { dac_override kill setgid setuid net_bind_service };
allow innd_t self:process setsched;
```

innd_t will need access to the listed domains. For the first allow rule below, innd_t will need to be able to search directories of types *bin_t* and *sbin_t*. For the second allow rule, innd_t will need read access to symbolic links of type *usr_t*. When you have a getattr operation, include a read operation as well (as in the third, fifth and sixth allow rules below) because when you require getattr access, you'll almost certainly be needing read access too.

```
allow innd_t { bin_t sbin_t }:dir search;
allow innd_t usr_t:lnk_file read;
allow innd_t usr_t:file { getattr read ioctl };
allow innd_t lib_t:file ioctl;
allow innd_t { etc_t resolv_conf_t }:file { getattr read };
allow innd_t { proc_t etc_runtime_t }:file { getattr read };
allow innd_t urandom_device_t:chr_file read;

allow innd_t innd_var_run_t:sock_file create_file_perms;
```

innd_t will need to be able to read the /etc/news directory and files in it. innd_etc_t is the type assigned to /etc/news (see innd.fc). The file global_macros.te contains the macro definition for *etcdir_domain*, so read it to get a better idea of what it does. To our innd.te policy file we now add

```
etcdir_domain(innd)
```

innd_t will need to create log files under /var/log with a type of *innd_var_log_t*. It will need a directory of its own that it can write to. To achieve this, we need

```
logdir_domain(innd)
```

innd_t will need read-write directory permissions to /var/lib/news so we include

```
var_lib_domain(innd)
```

Lastly, we need the crond_t domain to be able to transition to innd_t when executing a program with the type innd_exec_t.

```
ifdef('crond.te', '
system_crond_entry(innd_exec_t, innd_t)
')
```

13.2 the innd.fc file

The innd.fc file contains the file contexts we need to create for INN. These are the contents:

```
/usr/sbin/innd.*        --      system_u:object_r:innd_exec_t
/var/run/innd(/.*)?             system_u:object_r:innd_var_run_t
/etc/news(/.*)?                 system_u:object_r:innd_etc_t
/etc/news/boot          --      system_u:object_r:innd_exec_t
/var/spool/news(/.*)?           system_u:object_r:news_spool_t
/var/log/news(/.*)?             system_u:object_r:innd_log_t
/var/lib/news(/.*)?             system_u:object_r:innd_var_lib_t
/usr/sbin/in.nnrpd      --      system_u:object_r:innd_exec_t
/usr/lib/news/bin/.*    --      system_u:object_r:innd_exec_t
/usr/bin/inews          --      system_u:object_r:innd_exec_t
/usr/bin/rnews          --      system_u:object_r:innd_exec_t
```

On the lefthand side of the file, we have the files and directories specific to INN. On the righthand side we assign the contexts. It is best to follow naming conventions at all times, such as appending _exec_t to the daemon name for executable files. In the middle of innd.fc we have -- which refers to regular files only. The entry for /etc/news(/.*)? can not contain -- because that directory contains subdirectories and does not consist solely of regular files.

After saving your changes to this file, you need to apply the file contexts. From your policy source directory, run the following command to make the main file_contexts file so you can then relabel parts of the file system concerned with what you've specified in innd.fc :

```
make file_contexts/file_contexts
```

If you're not in the policy source directory, run

```
make -C file_contexts/file_contexts
```

The next step is to run the setfiles command to relabel each file or directory you have listed on the lefthand side. I'll take the first line of innd.fc as an example:

```
setfiles -v file_contexts/file_contexts /usr/sbin
```

After running this command, run a ls -Z command on /usr/sbin/innd.* to see that the relabelling has taken place.

13.3 The net_contexts file

The net_contexts file in your policy source directory contains security contexts for network entities. The following line needs to be added to this file so that you are assigning tcp port 119 the type innd_port_t and no other type will be able to bind to that port.

```
ifdef('innd.te', 'portcon tcp 119 system_u:object_r:innd_port_t')
```

---

14. Policy tools

A number of tools have been developed for SE Linux. Some are listed below, with links to the appropriate tools. I have not played around with them much myself, but please visit the sites of the authors for more information.

Tresys Technologies: SE Linux Policy Tools
Tresys have developed tools for the analysis of the SE Linux policy, a GUI and command line tool to assist with managing your SE Linux system, a GUI tool to browse and modify policy components, a policy debugging application, a tool for viewing policy statistics and a tool to search TE (type enforcement) rules. Please visit http://www.tresys.com/selinux/selinux_policy_tools.html for more information.

Mitre have developed a policy analysis tool, available at the NSA's SE Linux download site.

audit2allow was written by Justin R. Smith with contributions by Yuichi Nakamura and others not mentioned anywhere (apologies to those people). audit2allow takes the output of dmesg, analyses the avc denied messages and comes up with rules you can apply to fix those denied messages. It is included in the Debian package and Fedora rpm policycoreutils.

---

15. Resources

Please see the following links for more information.

NSA official site

The NSA's official SE Linux site is at http://www.nsa.gov/selinux

The NSA have published a document called Configuring the SE Linux Policy and is available at http://www.nsa.gov/selinux/papers/policy2-abs.cfm

The official SE Linux FAQ is at http://www.nsa.gov/info/faq.cfm

Additional NSA published papers, technical reports and presentations can be found at http://www.nsa.gov/selinux/info/docs.cfm

Mailing List and Archives

The NSA run a mailing list for discussion of all things SE Linux. Follow the instructions at http://www.nsa.gov/selinux/list.html to subscribe. The same URL describes how to obtain list archives.

Getting Started with SE Linux HOWTO: the new SE Linux

My document on how to get started with installing and using SE Linux can be found here.
