SELinux - an introduction

Tobias Hommel (tobias.hommel@rub.de)

January 19, 2009
ver. 0.1

This introduction was designed to help doing research work on SELinux. Otherwise it would take weeks to search and read the necessary documentation, if you haven't worked with SELinux before. This document is far from complete and only touches some subject areas of SELinux. It will try to explain the most important aspects needed to know when working the first time with SELinux. Although may sometimes being a bit superficial, it should still be correct.

Some examples in this document are taken from the book *SELinux by Example* [1], which is in my opinion the best printed book available on SELinux. Although it is not bleeding edge, like all the other books out there it covers many aspects of SELinux.

There is no part about installing a new SELinux system or converting a traditional system to SELinux in this document. Although this might be important, I think it doesn't help much in understanding SELinux. If you need help on setting up an SELinux system, refer to the documentation of your linux distribution. The Gentoo SELinux Handbook gives a nice tutorial on how to convert a system. [3]

Contents

1 Introduction

In 2000 the NSA released SELinux to the open source community. At this time it was a bunch of kernel patches that implemented the FLASK [1] architecture. FLASK [5] is an operating system architecture that was originally implemented in the Fluke operating system and demonstrates mandatory access control. Fluke is a research operating system developed by the Flux Research Group at Utah University.

To make SELinux part of the official Linux kernel some effort had to be made first. The chief maintainer of the linux kernel, Linus Torvalds, told that it would not become part of the main kernel until some security framework had been integrated. This was the birth of the LSM interface.

LSM is a security framework that is part of the Linux kernel since the 2.6 series. It is integrated into important kernel functions to control access to objects like files, network interfaces and more. SELinux was ported to the LSM interface and became part of the Linux kernel.

The first linux distribution that came up with SELinux support was Fedora Core, that was sponsored by Red Hat. Today it is more or less supported by other distributions, amongst others are of course Fedora and Red Hat, short after those came Gentoo, many others followed, e.g. Debian.

1.1 Motivation

So why do we need SELinux? Is the classic discretionary access control system not sufficient? Let's take a look at a linux system:

```
1  $ find /bin -perm +4000 -uid 0 -exec ls -l {} \;
2  -rwsr-xr-x 1 root root 22064 2008-09-25 18:06 /bin/fusermount
3  -rwsr-xr-x 1 root root 31012 2008-11-13 07:54 /bin/su
4  -rwsr-xr-x 1 root root 92584 2008-09-25 15:08 /bin/mount
5  -rwsr-xr-x 1 root root 30856 2007-12-10 18:33 /bin/ping
6  -rwsr-xr-x 1 root root 26684 2007-12-10 18:33 /bin/ping6
7  -rwsr-xr-x 1 root root 71556 2008-09-25 15:08 /bin/umount
```

Listing 1: suid root programs

This lists all the files in `/bin/` that are owned by user *root* and have the setuid flag set. The setuid flag tells the system that, whoever executes the file runs this file with the permissions of the owner. That means in our case, if you use mount to e.g. mount an usb stick, you run this command as user root. The problem with this situation might be obvious: if someone manages to find a bug in one of these setuid-programs, he might be able to get root access to the system and may execute privileged commands.

So what can SELinux do about this? – Short answer - the principle of *least privilege*. You can control what kind of actions ping is allowed to perform on an object. You can for instance say the process that results from executing the file `/bin/ping` can open raw sockets to perform it's main task. It also should have access to the DNS system to

[1] **F**lux **A**dvanced **S**ecurity **K**ernel

resolve hostnames. Every other action should be denied, e.g. read or write access to files. We will see how this works in detail later on.

But what if you have an application that has to be run as root and has to write to files, like a syslog daemon? This application can of course write to every file on a system, even on ones it should not. On traditional linux systems this is correct. On SELinux you can run at-risk programs in their own domains. Then you can only allow access to special resources for this domain.

So one of the main problems in traditional linux is the role of the root user, who can normally do everything on a linux system. But there are other problems as well. Discretionary access control in linux terms means that the owner of a resource has the exclusive power over it. On the one hand this is O.K., because you, as the owner of your document, can choose who should have access to it. On the other hand this can also lead to problems, as not all users are aware of the problems that weak file permissions may bring.

2 SELinux Terms

To understand SELinux we have to explain some important commonly used terms first.

- **subject** – A *subject* is the process that wants to access an *object*.
- **object** – When talking about an *object* we mean a resource like a file or a network interface. But also another process may be an object, or even the subject itself may be the object (self). Each object has an *object class*.
- **object class** – Each object belongs to an object class. There are 4 kinds of object classes in SELinux:
  - file related objects - e.g. `dir`, `file`, `lnk_file`, `sock_file`
  - network and socket objects - e.g. `socket`, `netif`(network interface) or more specific types like `tcp_socket`
  - IPC [2] related objects - e.g. `msg`, `shm`
  - miscellaneous - e.g. `capability`, `passwd`

  A list of available object classes can be found in the documentation of the reference policy by Tresys. [8] Although object classes can be defined, as a list of permissions, with the **class** statement, this makes only sense if the newly defined class is supported by the kernel.
- **Security Context** – Each *subject* and *object* in SELinux have a *security context*. It basically consists of 3 attributes: a *user*, a *role* and a *type*. If the MLS/MCS (multi-level-security/multi-category-security) *policy* is used, there is also an attribute for the range/category.

[2] **i**nter-**p**rocess **c**ommunication

```
1  SEStrict ~ # ls -Zl /
2  total 48
3  drwxr-xr-x+  2 root root system_u:object_r:bin_t          4096 Nov
       10 14:04 bin
4  drwxr-xr-x+  3 root root system_u:object_r:boot_t         4096 Nov
       26 04:29 boot
5  drwxr-xr-x+ 10 root root system_u:object_r:device_t      13040 Nov
       26 05:38 dev
6  drwxr-xr-x+ 59 root root system_u:object_r:etc_t          4096 Nov
       26 05:37 etc
7  drwxr-xr-x+  3 root root system_u:object_r:home_root_t    4096 Oct
       28 23:15 home
8  drwxr-xr-x+  8 root root system_u:object_r:lib_t          4096 Nov
       10 10:08 lib
9  drwx------+  2 root root system_u:object_r:lost_found_t   4096 Oct
       28 22:44 lost+found
10 drwxr-xr-x+  4 root root system_u:object_r:mnt_t          4096 Jun
       11 19:46 mnt
11 drwxr-xr-x+  3 root root system_u:object_r:usr_t          4096 Nov
       14 12:04 opt
12 dr-xr-xr-x+ 51 root root system_u:object_r:proc_t            0 Nov
       26 05:37 proc
13 drwx------+ 17 root root root:object_r:sysadm_home_dir_t  4096 Nov
       27 09:00 root
14 drwxr-xr-x+  2 root root system_u:object_r:bin_t          4096 Nov
       26 05:34 sbin
15 drwxr-xr-x+  7 root root system_u:object_r:security_t        0 Nov
       26 05:37 selinux
16 drwxr-xr-x+ 11 root root system_u:object_r:sysfs_t           0 Nov
       26 05:37 sys
17 drwxrwxrwt+  4 root root system_u:object_r:tmpfs_t          80 Nov
       27 03:10 tmp
18 drwxr-xr-x+ 14 root root system_u:object_r:usr_t          4096 Oct
       29 21:15 usr
19 drwxr-xr-x+ 13 root root system_u:object_r:var_t          4096 Oct
       26 18:15 var
```

Listing 2: some default security contexts

In listing 2 we already get a little impression of the importance of the 3 portions of a security context, at least for objects.

- **Labeling** – Files and directories on a filesystem are labeled with a *security context*.
- **SELinux User** – The *SELinux user* has nothing to do with the standard system user. Actually there are several system users sharing a single *SELinux user* on an SELinux system. The task of the *SELinux user* is to decide which *roles* a system user is allowed to take.

  SELinux users are typically in the form `name_u`. The `_u` suffix is no must, it is just conventionally used for SELinux users. Some typical users are `root`(which is

not suffixed with `_u`), `system_u` which is used for all kinds of system processes, e.g. init.

- **Role** – The *role* is used to determine which *domains* a process can enter. It is absolutely unimportant for *objects* and only exists to make the security context complete.

  SELinux roles are, like SELinux users, suffixed. Yes, right, with `_r` of course. Here, the suffix is for traditional purpose and optional, too. Some standard roles, that exist on most SELinux systems are `user_r`, which is normally used for common users. The `staff_r` role is similar to the `user_r` role, with the difference, that a member of the `staff_r` is allowed to change to the `sysadm_r` role. The `sysadm_r` role is the role, that gives back most of the powers to root, that other roles, like `staff_r` forbid. Objects always take the special role `object_r`, which is hard-coded. `object_r` must **never** be defined or used in any way, this could be a security issue, since a process with this role could potentially enter any domain.

- **Type** – Each *object* in SELinux has a *type*. In the *policy* it is used to determine which *subjects* can access the *object*. The *type* is the most important portion of a security context in SELinux and you'll find many different types.

  As you might already have guessed, SELinux types are typically in the form `type_t`.

- **Domain** – When talking about the *type* of a *process*, we call it *domain*. Technically speaking there is no difference between *domain* and *type*. Access to a domain, as everything in SELinux, has to be allowed explicitly. A process can only enter a specific domain on execution.

- **Permissions** – The *permissions* define in which way a *subject* is allowed to access an *object*. There are several types of *permissions* in SELinux that we know from traditional linux, like `read` and `write`, but also some finer grained ones like `append` or `getattr`. This doesn't suffice for all objects of course, so there are object class specific permissions, like `create` or `bind` for sockets, or `add_name` to create or move a file to a directory. A list of available permissions can be found in the documentation of the reference policy by Tresys. [8]

- **Access Vector** – An *access vector* is a set of permissions that a subject is allowed to perform on an object.

- **Alias** – An alias can be defined for types. This makes sense, if the policy changes, for compatibility reasons. Remember Netscape? It was somehow the predecessor of Mozilla. To make the types `mozilla_t` and `netscape_t` mean the same, we can define the latter as an alias.

```
1  #method 1
2  type mozilla_t;
3  typealias mozilla_t alias netscape_t;
4  #method 2 (same as method 1)
5  type mozilla_t alias netscape_t;
```

- **Attribute** – Attributes can be used to group types.

```
1  # declare the attribute config_file
2  attribute config_file;
3  # assign types to attribute (method 1)
4  type etc_t;
5  type shadow_t;
6  typeattribute etc_t config_file;
7  typeattribute shadow_t config_file;
8  # assign types to attribute (method 2)
9  type etc_t, config_file;
10 type shadow_t, config_file;
11
12 #use the attribute in an allowrule
13 allow sysadm_t config_file:file read;
```

  In the example above we use the attribute to make an allow rule. Without the attribute, we would have had to write the rule for both types. An attribute is internally expanded to the list of types that have this attribute.

- **Policy** – We will use the term *policy* to describe a set of *rules* that define if and how a *subject* can access an *object*.

- **Relabeling** – An SELinux policy can allow trusted *processes* to change the label of an *object*. Examples are, if you login, the program login changes the label of the controlling tty. The *permissions* that are related to type changes are `relabelto` and `relabelfrom`.

- **Type Changes** – You can specify rules in the policy, that tell SELinux-aware applications, how they should relabel a file.

```
type_change sysadm_t tty_device_t:chr_file sysadm_tty_device_t;
```

  This rule tells an SELinux-aware process in the `sysadm_t` domain, that a file of object class `chr_file` should get the type `sysadm_tty_device_t` when relabeling it.

- **Type Enforcement** – If you watch the three definitions of *SELinux user*, *Role* and *Type* carefully, you maybe have already seen the importance of each. The most important one is the *type*. It is used to define what *permissions* a *subject* of a certain source *type* is allowed on an *object* of another certain target *type*. This technology is called Type Enforcement™ (TE). Type Enforcement is a registered trademark of Secure Computing. An interesting thing to mention is probably the Statement of Assurance [10] from Secure Computing regarding SELinux.

- **Role-Based Access Control** – To determine which *permissions* a system user has on a certain *object*, he has to have access to a certain *domain* that is allowed to access the specified *object*. To group users who should have common access rights, a *role* is defined. Each user who can change to this *role* can then *transition* to the

needed *domain*, if allowed. It is easier to maintain a *role* than to give each user the same rights. It is important to notice, that this Role Based Access Control (RBAC) is not really an access control mechanism, since it only further constraints type enforcement, but doesn't allow access itself.

- **User-Based Access Control** – UBAC is a really new mechanism in SELinux, since the reference policy version 2.20081210. It is based on constraints, that apply to the user portion of both involved security contexts, the one of the accessing process and the one of the object. To allow access to constrained objects, both user portions must be equal. UBAC is no replacement for RBAC, but can additionally be enabled. If you read anything like "UBAC is a replacement for RBAC", don't get confused. They don't mean RBAC in the meaning of *Role-Based Access Control* we defined above.

3 How it Works

3.1 LSM

The SELinux we are talking about is implemented as an LSM(Linux Security Modules) module. [11] Roughly speaking the LSM architecture is a set of functions that hook into the kernel code. Those hooks are called in the kernel before critical operations are made, such as file or network access or operations on processes or IPC. Those functions are called whenever a permission check is to be made. Unlike the classic DAC checks, the LSM hooks are applied on **every** access. So on a traditional linux system you can change the permissions of a file while a process has opened it and the process won't notice this while it stays open. If you do this on a system with an active LSM module, the file might not be accessible by the process anymore, depending on the rules. In

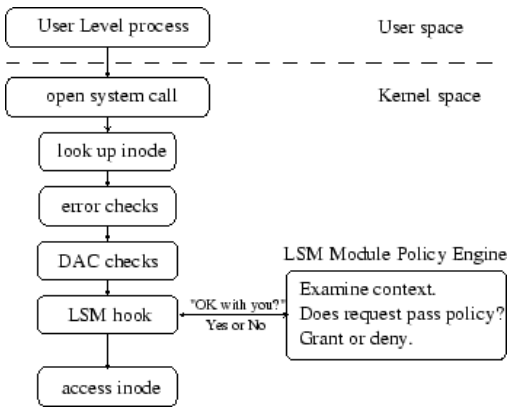

Figure 1: LSM Hook Architecture

Figure 1 you can see that the LSM hooks are called after the DAC checks. LSM hooks are normally *restrictive*, that means, they can deny access that has been allowed by the kernel, but cannot allow access that has been forbidden by the kernel. So you can never give more privileges to a process with SELinux than it would have without it. There are also *permissive* LSM hooks, they are needed for POSIX capabilities, which are also implemented as an LSM module since the 2.6 kernel series.

Besides SELinux there are some other LSM modules, e.g. AppArmor or LIDS, to mention some well-known ones.

3.2 The SELinux Architecture

As already mentioned, SELinux is based on the FLASK architecture. The whole construct is split into a policy-enforcing part and decision-making part. So we have a strict separation between policy enforcement and policy decision-making here. The policy enforcement part is the *object manager* that forbids or allows access. In fact it is more precise to say the *object manager**s***, because we have actually more than one, at least in SELinux. In SELinux the *object managers* are the LSM hooks, that are scattered throughout the kernel code. The policy decision-making part consists of the *security server* which makes the decision if an action is allowed or not. For better performance there is the *Access Vector Cache*(AVC), that caches decisions made by the security server. If the object manager gets a request, it gets the security contexts of the subject and the object. It then converts the security contexts into SIDs (security identifier) and sends those two SIDs together with the object class to the AVC to ask for a decision. If the AVC doesn't have a decision cached, the AVC asks the security server. The security server checks the binary policy that is loaded into the kernel and makes a decision. This decision will be made in two steps. First the security server generates a mask of the permissions allowed by rules. Then it will remove allowed permissions, that are disallowed by a constraint. The decision, in the form of an access vector, is then sent back to the AVC, which stores it for future requests and sends it back to the object manager. Based on the reply, the object manager allows or denies access.

SIDs are only used internally. In userspace, only security contexts are used. The security server maintains a table that maps SIDs to security contexts.

Since it is not unlikely, that you enhance or change the policy, it would really be unacceptable if you have to restart the whole system for each change you make. So, of course, you can load a new policy into the running kernel. If the policy gets changed in any way, of course the security server is noticed, since it makes the decisions. It will then update its SID mappings and with it reset the AVC.

3.3 SELinux Files and Filesystems

On SELinux there are two important locations. One is the configuration directory `/etc/selinux`, the other is the selinux filesystem mounted at `/selinux`.

[2] image in Figure 1 taken from `http://www.usenix.org/event/sec02/full_papers/wright/wright_html/node3.html`

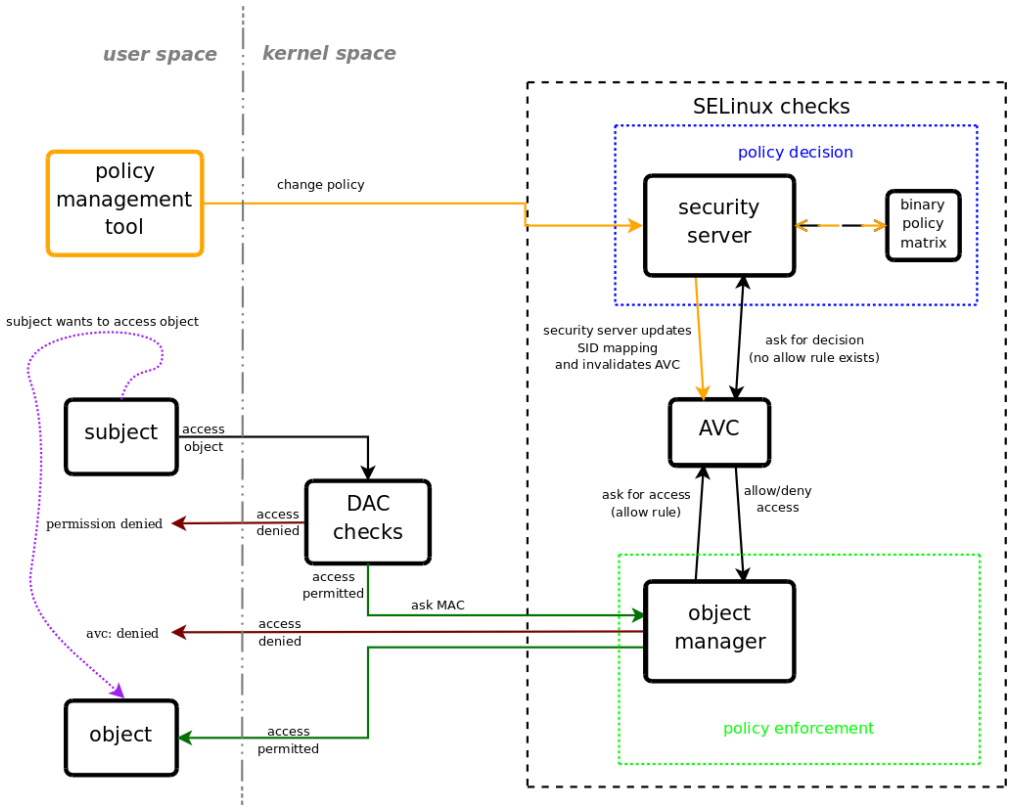

Figure 2: selinux architecture

3.3.1 /selinux

This is a pseudo filesystem like `/proc` or `/sys` and contains information about the current state of the SELinux system. There are several "files" that can be used to get or set states and values of the system. Here are some examples:

- `/selinux/enforce` – contains the enforcing state
- `/selinux/booleans/` – contains a file for each Boolean on the system
- `/selinux/class/` – contains subdirectories for each object class (file, dir, socket, ...) and the available permissions

3.3.2 /etc/selinux

This directory contains SELinux specific configuration and the binary policy. The file `/etc/selinux/config` contains the state of the SELinux system on boot. It contains the policy type and the enforcement mode.

There is also a directory for each policy, e.g. for the strict policy `/etc/selinux/strict/`, that contains subdirectories, that in turn contain the binary policy, the active policy modules and information about SELinux users and file contexts.

3.4 File Labeling

Each object and subject in SELinux has to be labeled correctly. A process (subject) simply inherits the security context of its parent, but what about files? In SELinux there are 4 mechanisms for labeling a file-related object.

- labeling based on extended attributes (xattrs)
- task-based labeling
- transition-based labeling
- generalized labeling

If a filesystem supports extended attributes, the security context is saved there. Examples for those filesystems are *ext2/ext3*, *xfs* and *reiserfs*. If a file-related object on such a filesystem gets labeled, it normally inherits the user portion of the security context from the creating process. The role portion is always `object_r`. The type portion is inherited from the directory containing the file or, if there exists a type_transition rule for the domain of the process and the type of the directory containing the file, the type specified by this rule is used.

The task-based labeling is for instance used if you use pipes, the security context is inherited from the creating process.

Transition-based labeling is similar to the task-based labeling, but it uses type_transition rules for the labeling, like the xattrs-based labeling. It is for example used on *tmpfs*, *shm*, or *devpts* filesystems.

Generalized labeling is used for pseudo filesystems like *proc*, *sysfs* or *selinuxfs* and for filesystems that don't support extended attributes like *fat* or *iso9660*.

```
1  fs_use_xattr          ext3            system_u:object_r:fs_t;
2  fs_use_task           pipefs          system_u:object_r:fs_t;
3  fs_use_trans          tmpfs           system_u:object_r:tmpfs_t;
4  genfscon              proc    /kmsg   system_u:object_r:proc_kmsg_t
```

Listing 3: filesystem labeling statements

If you want to change the default label for unlabeled files, the command mount supports the options **context=**, **fscontext=** and **defcontext=** on SELinux.

Other objects like network interfaces or ports are also labeled, but the rules are similar to those of file-related object labeling.

3.5 Enforcing, Permissive or Disabled?

You can, if enabled in the kernel, run SELinux in two modes, permissive and enforcing. The default mode can be set in `/etc/selinux/config` and can be overridden by setting

the kernel boot option `enforcing` to 0 or 1. You can also get and set this state with the command `getenforce` and `setenforce`.

- enforcing – The policy is enforced, access is allowed based on the policy.
- permissive – The policy is not enforced, but log messages are generated as in enforcing mode.
- disabled – This is not a real mode like the previous two, it is just listed here to mention, that SELinux can be disabled by setting the kernel boot option `selinux` to 0. If configured in the kernel this can also be done at runtime.

3.6 The Boot Process on SELinux

The boot process on SELinux is quite interesting, since it is some kind of chicken and egg issue. On an SELinux system each process has to be labeled with the correct domain. This also applies to **init**, the first process started on a linux system. To label a process correctly, the policy has to be asked.

The question now is, how can a process be labeled before the policy is loaded? – Here is a list of the boot process on an SELinux system:

1. The bootloader loads the linux kernel.
2. The kernel loads init and assigns the predefined *initial SELinux ID* to it.
3. **init** mounts `/proc` and looks for the **selinuxfs** filesystem type.
4. If SELinux is disabled the system boots like a traditional linux system.
5. If it is enabled, `/selinux` is mounted.
6. Now init checks which policy version is supported by checking `/selinux/policyvers`.
7. **init** now checks `/etc/selinux/config` for the type of policy that should be active and loads the policy from `/etc/selinux/$POLICY_TYPE/policy.$POLICY_VERSION`. If the policy version is not supported by the kernel, an older version is loaded.
8. If there are Booleans, that don't have their default value, they are set and the policy is modified in the memory.
9. The kernel does not enforce any policy until the initial policy is loaded. After it has been loaded, the enforcing mode(enforcing, permissive) from the config or the kernel commandline is "active".
10. **init** now re-executes itself and transitions to its correct domain based on the policy.
11. The boot process continues as it would without SELinux.

3.7 Domains, Types and Transition

Now let's see what happens if you use a simple command like `/bin/ls`.

At first the classic DAC system checks if you are allowed to execute the file `/bin/ls`. 1 At least that is the case if `/bin/ls` is a binary. If it is an interpreted script(perl, bash, python, ...) you will also need read permission. If you are not allowed to execute the file the procedure ends here. You just can't execute the file. If the DAC permits access, the next part is to ask the MAC system.

The MAC system checks if a rule exists that allows the domain of your shell process to execute `/bin/ls`. If this is the case, `/bin/ls` gets executed. If there is no allow rule you can't execute the command.

Since `/bin/ls` is a command that isn't really a potential security risk, let's see what happens if you execute `/bin/passwd`. This is a bit more tricky, since a bug in the program could have disastrous impact on the security of a system. Take a look at the following situation:

```
1 SEStrict ~ # id -un
2 root
3 SEStrict ~ # id -Z
4 root:sysadm_r:sysadm_t
5 SEStrict ~ # cat /etc/shadow
6 cat: /etc/shadow: Permission denied
7 SEStrict ~ # ls -lZ /bin/passwd /etc/shadow
8 -rws--x--x+ 1 root root system_u:object_r:passwd_exec_t 32844 Oct 28 ↷
      15:55 /bin/passwd
9 -rw--w----+ 1 root root system_u:object_r:shadow_t        694 Dec 11 ↷
      14:43 /etc/shadow
```

Listing 4: even root in the sysadm_r role can't directly read /etc/shadow

In listing 4 we see that even the almighty root user can't read the file `/etc/shadow`. Even more interesting is, that he's in the `sysadm_r` role, which is normally the most powerful role on SELinux systems. So why can the program `/bin/passwd` access `/etc/shadow`? Since Type Enforcement is the mechanism in SELinux that defines what access is allowed, we'll take a closer look at this and scan through the source of our policy. Obviously there does not exist a TE rule that allows the `sysadm_t` domain to access objects of the type `shadow_t`.

Since SELinux denies every access by default, obviously there have to be some rules that allow the access. Of course you will find some rules. Those allow `passwd_t` to access `/etc/shadow`, but not `sysadm_t`. So, why can `/bin/passwd` read `/etc/shadow` anyway, even though you are running it from the `sysadm_t` domain?

If a program is executed and a process is generated, this process inherits the security context of the calling process. So if your current bash session has the security context `user_u:user_r:user_t`, it runs in the `user_t` domain. If you execute a command, it will also run in the `user_t` domain. But if you need to define special rules for a program like `passwd`, you somehow have to make those rules unique to this one program.

Let's see in detail what exactly happens when you call `/bin/passwd`. Here, too, the DAC will be applied first, of course. The difference lies within the MAC part. The

first thing is to check whether you are allowed to execute the file `/bin/passwd`. If you are allowed to execute the file, it is executed. If your shell process runs in the domain `user_t`, an appropriate allow rule could look like this:

```
allow user_t passwd_exec_t:file { getattr execute };
```

Now comes the special part. The keyword here is *domain transition*. Domain Transition happens when a process changes from its old domain to a new domain.

If you start a program (e.g. `/bin/passwd` or `/bin/ls`), the process that is being created will inherit the security context of your shell. That's fine for `/bin/ls`, but not enough for `/bin/passwd`, since we want special security. So we also have to create some special rules, that is, we need to run passwd in a special domain to apply custom rules to it.

To let passwd run in a unique domain (`passwd_t`) we need another domain that helps us with the transition to this domain, since we cannot enter `passwd_t` directly. What we do is, we label the file `/bin/passwd` with the type `passwd_exec_t`. Then we give the domain `passwd_t` the special permission `entrypoint` on objects with the type `passwd_exec_t` of class `file`.

```
allow passwd_t passwd_exec_t:file entrypoint;
```

This rule defines which files are allowed to enter the domain `passwd_t`. That is still not enough. This rule defines something like a door, but a door with a bouncer. To get past a bouncer you need special permission, like an entrance card or being female with long blonde hair or just being stronger than the bouncer. Since your shell process is genderless and is normally not stronger than the enforcement server of the system, it needs a ticket. The ticket for your shell process looks like this:

```
allow user_t passwd_t:process transition;
```

Let's sum up. We have `/etc/shadow` of type `shadow_t`, that can only be read by a process in domain `passwd_t`, which again can only be entered by executing a file of type `passwd_exec_t`.

We have still one last, but important problem left. All rules necessary to transition and to execute are ready, but how does `/bin/passwd` actually transition to the correct domain if it is executed? If passwd were an SELinux-aware program, it could do this by itself now. Since it isn't and we want to cover the case for every command, we need a fourth rule to make the system manage the transition automatically.

```
type_transition user_t passwd_exec_t:process passwd_t;
```

This only makes a process created from a file of type `passwd_exec_t` transition to `passwd_t` automatically if it is created by a process in the domain `user_t`. The previous allow rules are still needed.

So why do we need this amount of effort? If `/bin/passwd` is the only file of type `passwd_exec_t` on a system, it is obviously the only way to access `/etc/shadow`. So no other process will ever be in the position to access it. If you find a bug in `/bin/passwd`, you may of course be able to do damage to `/etc/shadow`, but this lies beyond the powers of SELinux and is a problem that can only be fixed by a passwd developer.

3.8 Roles

When we talked about domain transition in the last part, we left out one thing completely. The RBAC part of SELinux. Although roles are not the primary access control mechanism, they can be used to further limit the use of domains. Since roles are assigned to SELinux users, a role defines which domains can coexist together with which user-portion.

So for the `/bin/passwd` program to work, we actually need a fifth rule. A rule that makes the security context `user_u:user_r:passwd_t` valid.

```
role user_r types passwd_t;
```

For roles there also exist role transitions, but they are not as frequently used as domain transitions. A role transition is for example needed when a daemon is started by a system administrator.

```
role_transition sysadm_r daemon_exec_t system_r;
```

This rule states that if a process with the role `sysadm_r` executes a file with the type `daemon_exec_t`, the role should be changed to `system_r`, so that the daemon with this type always runs with the same role.

These role transitions must be allowed, of course. There is another allow statement in SELinux that applies to roles.

```
allow sysadm_r system_r;
```

Although this role allow statement is similar to the type allow rule, note that they are two different things.

3.9 Optional Policies

Since the development of the reference policy you don't have to compile the whole policy source, but can split it up into modules. This gives flexibility but also the need for attention when one module depends on another module. If there is such a dependency, the needed module must be loaded to load the module that needs it.

This is O.K. if the existence of the needed module is mandatory. If your module does not necessarily depend on another module, but only supports it, you have the option to declare a portion of the rules in an optional-block.

```
1 optional {
2   apache_read_log(fail2ban_t)
3 }
```

This is a slightly modified example from the policymodule of fail2ban, a script that analyzes logfiles and acts based on the entries. If all the types, roles, permissions, etc., that are used in the optional block are defined somewhere in the loaded policy, the policy statements in the optional-block are loaded, too. Now, every time a module gets loaded or unloaded, this optional policy is checked and the statements are enabled or disabled accordingly.

3.10 Booleans

If a policy rule does not meet your needs, or you prefer a different behaviour, luckily you can change the policy and load it into the running kernel. Even though this is better than rebooting the system for every change, it might sometimes be helpful if you wouldn't have to mess around with writing policy, analyzing logfiles, mediate on the need for a special rule and such.

For some rules exist booleans. Booleans in SELinux are variables you can set to change the behaviour of the policy on-the-fly. They make it possible to change the behaviour of the policy without reloading it. You can get a list of available booleans with `getsebool -a`.

```
1  bool user_ping false;
2  # ...
3  if (user_ping) {
4    #allow ping for regular users
5  } else {
6    #optionally do something else
7  }
```

Listing 5: definition and usage of a boolean variable

In the policy a boolean is defined with the bool statement, then follows the name of the boolean and at last the default value, which must be **true** or **false**.

To use the boolean in the policy we use the **if**-statement, which evaluates the expression in brackets after the **if** and applies rules based on this. Not every kind of rule is allowed in this conditional statement, e.g. role allow rules. Also nesting of if-statements is not allowed, but there are several boolean operators one can use that are well-known from some programming languages.

3.11 Dominance

A role can be defined to dominate other roles. This is can be achieved with the **dominance** statement.

```
dominance { role uber_r {role sysadm_r; role secadm_r}; }
```

If a role dominates other roles, it inherits all their type associations. In the example above, `uber_r` will gain all access that the other two roles had before the dominance statement. Every type that is associated with, e.g. `sysadm_r`, after this doesn't affect the `uber_r` role. A dominance can also be used in the **constrain** statement to express a relation between two roles. There are 4 dominance relationship operators.

- *dom* – r1 was defined to dominate r2
- *domby* – r2 was defined to dominated r1
- *eq* – the same as ==
- *incomp* – both roles don't dominate each other

Dominance is also important for MLS, but a bit different of course.

3.12 Constraints

Normally all access in SELinux must be allowed. This is true and will stay true, but there are mechanisms that can be used to disallow an allowed permission. More precisely spoken, it will constrain the allowance. In SELinux this can be done with the **constrain** statement.

```
constrain process transition ( u1 == u2 );
```

This rule will constrain the `transition` permission for the objectclass `process`. It will only allow `transition` on a process if the user portions of both security contexts are equal. The keywords **u1** and **u2** indicate, respectively, the user portion of the security context of the subject and the object. There are also the keywords **r1**, **r2**, **t1** and **t2**, which apply to the role and type portions of the subject and object. Instead of the == operator there can also be the **!=** operator or one of the dominance operators, if we compare roles. The left side of the comparison always has to be a keyword and the right side can be a keyword or an identifier.

Another constrain mechanism is the `validatetrans` statement. It can be used to further control the ability to relabel an object. It is only mentioned here for completeness and won't be discussed further since it is not used in the reference policy.

4 The Policy

Let's have a look at the SELinux policy now. There are many kinds of access on a system, but normally a program needs only few of them. The normal behaviour of SELinux is to deny every access and a good rule should only allow the kind of access for a process that it really needs.

4.1 The Reference Policy

There are several programs that run on every Linux system, so it wouldn't make sense to reinvent the wheel every time we install an SELinux system. Fortunately there are basic rules for commonly used programs, such as mount, ifconfig, etc. These standard rules make the base policy.

In the beginning there was the example policy by the NSA. It was a set of rules for programs that exist on nearly every linux system. Today the reference policy [6] has replaced the example policy. It is based on the example policy and is maintained by Tresys Technologies.

Some of the main development goals of the reference policy were:

- one policy source for all policy variants (strict, targeted, MLS)
- modularity
- integration of documentation into the policy source
- make it easier to maintain the policy (m4 macros)

4.2 Compiling the Policy

If you have ever compiled a linux kernel by yourself, you will probably know, that you can compile most things either directly into the kernel or you can build them as separate modules. Building a policy is somehow like building a kernel.

First of all you will probably download the reference policy source, which already contains rules for most things. The reference policy is split up into several policy source modules. There are source modules for many tasks, e.g. applications like Mozilla or daemons like distcc, but also for roles like `sysadm_r`. To make the needed rules usable by the kernel, they have to be compiled into binary form. This is done by the policy compiler `checkpolicy/checkmodule`. The created modules have the extension `.pp`. You will most certainly never have to use the policy compiler directly, but you will use a makefile instead, that takes care of everything.

You can build either a monolithic policy, which will be one big module. Or you can build a modular policy, which will consist of a base module and optionally some loadable modules. Building a modular policy has the advantage, that you don't need to rebuild the complete policy for every little change you make. You can load modules for the programs, as you need them.

If compiling a modular policy, the base module will contain the most important source modules. Most important is normally defined as those rules needed to boot the system correctly. This depends on you linux distribution and configuration, of course.

To finally activate or deactivate all modules, you use the `semodule` command.

4.3 Type Enforcement Rules

To actually write policy, we need to have a look on the syntax and different parts of a policy.

We have Type Enforcement, so obviously we need to define types. In line 4 in listing 6 for example, the type `user_t` is declared (this is an example, the type `user_t` is probably already defined on each system).

```
1  require {
2    type shadow_t;
3  }
4  type user_t;
5  require {
6    type etc_t;
7    type bin_t;
8  }
9  allow user_t etc_t:file read;
10 allow user_t bin_t:file { execute getattr read };
11 dontaudit user_t shadow_t:file getattr;
12 neverallow user_t shadow_t:file read;
```

Listing 6: some type enforcement rules

Note, that everything, that is used in a policy module has to be declared somewhere, this includes types, booleans, attribute, permissions, ... This is either be done using

the proper declaration statement, like the `type` statement in listing 6 or, if the used type is declared somewhere else in the loaded policy, by using the `require` statement. It doesn't need to be specified where exactly the required statement can be found, only that it is already defined. There can be more than one require block. If you need a require block in an interface file, you should use the predefined macro **gen_require()**.

An actual type enforcement rule, or access vector rule, is made up of five parts.

- rule type – this can be one of

  `allow` which indicates an allow rule

  `dontaudit` which does not allow something, but only supresses the log message

  `auditallow` which allows and additionally generates a log message

  `neverallow` is used to prevent careless policy writing, if a neverallow rule is defined for a type and an allow rule, this will give an error when compiling the policy.

- source type – is usually the domain of a process that wants to access something
- target type – is the type of the object the source wants to access
- object class – the class of the target, e.g. `file` or `socket`
- permissions – a set of permissions the source is allowed to the target

If you look at listing 6 you can see some access vector rules. The rule in line 7 allows a process in the domain `user_t` to read files of type `etc_t`. Line 8 contains an allow rule that allows more than one permission. The **dontaudit** rule says that no log message should be generated if a process in domain `user_t` tries to get the attributes of a file of type `shadow_t`. If you, for example, use the command `ls -lZ /etc/shadow`, **getattr** is needed to show the owner, date and other object information. The **neverallow** rule makes it impossible to define an allow rule that would allow read access for `user_t` to `shadow_t`.

There are many other things to know when talking about Type Enforcement rules, but we won't go into more detail here. Just one last interesting thing. The special type **self**. If you find a rule, e.g. `allow type_t self:process signal;`, this means, that a process in domain `type_t` is allowed to send itself a signal (except SIGKILL and some other critical signals).

4.4 Strict and Targeted Policy

The basic concept of SELinux is to deny every access and only allow access for explicitly defined actions. You will have this behaviour if you use the **strict** policy type.

For desktop systems this doesn't make much sense, since you will install, upgrade and deinstall many programs in the lifetime of this system that may need different policy rules from one version to another. To make life a bit easier with SELinux but still secure, the **targeted** policy was invented. In the targeted policy there is a special

domain that all processes run in unless specific rules have been made. Processes that run in this unconfined domain work as expected in most cases. Applications that need special attention, mostly network daemons, but also client applications will be confined as in the strict policy. So from the technical point-of-view, SELinux still denies every access.

You get both types of policy from the reference policy.

4.5 Multi-Level-Security

SELinux also implements multi-level-security (MLS), which can optionally be enabled when compiling the policy. MLS is an extended implementation of the Bell-LaPadula(BLP) model. If enabled, MLS constraints checks must be passed together with the TE rules to permit access.

```
1  # declare sensitivities
2  sensitivity s0;
3  sensitivity s1;
4  # declare an aliased sensitivity
5  sensitivity s2 alias high;
6  # specify the hierarchy of our sensitivities from "low" (s0) to "high
       " (s2)
7  dominance { s0 s1 s2 }
8
9  # define categories with aliases(optional)
10 category c0 alias red;
11 category c1 alias green;
12 category c2 alias blue;
13
14 # define allowed security levels
15 level s0:c0.c2;
16 level s1:c0.c1;
17 level s2:c0,c2;
```

Listing 7: MLS rules

In listing 7 we define sensitivities, categories and allowed security levels. The category and sensitivity statements just define and alias categories and sensitivities. The dominance statement defines the hierarchical relationship between our sensitivities. The level statement defines which combinations are allowed. Notice the **,** and **.** between the categories in the level statements. **.** marks a range (in the order we defined the categories), **,** lists categories.

Taken the definitions in listing 7, we now have a look at security contexts. When using an MLS-enabled policy, the security context is extended with two additional fields, the *low* (*current*) and *high* (*clearance*) security level, which, in turn, consist of a sensitivity and a set of categories. The high level of a valid security context must always dominate the low level. Sensitivities are **hierarchical** and can be compared using equivalence relationships($<$, $=$, $>$). Security levels ar **non-hierarchical** and can be compared using a dominance relationship.

The *low* level indicates the *current* security level of a process or the sensitivity of the data an object contains. The *high* level indicates the *clearance* of the user portion of the security context (so it determines the highest possible security level allowed for the *current* level of any security context).

```
1  # valid contexts
2  user_u:user_r:user_t:s0-s2:c0
3  user_u:user_r:user_t:s0:c0-s2:c2
4
5  # invalid contexts
6  user_u:user_r:user_t:s0:c0-s0:c2 # the high level doesn't dominate
       the low level
```

Listing 8: security contexts in an MLS policy

To make use of the security levels in the policy, you would use the **mlsconstrain** and **mlsvalidatetrans** statements, which are based on the **constrain** and **validatetrans** statements. The difference is that the MLS variants can only be used with an MLS-policy and the normal variants in both policies. The **mlsconstrain** statement has the additional keywords **l1**, **h1** and **l2**, **h2** which apply to the current security level of the subject and current security level of the object, respectively. Another difference is that dominance operators can be used in conjunction with security levels (sl1 and sl2). They are a bit different from the dominance operators for roles.

- *dom* – sl1 dominates sl2, if the sensitivity of sl1 is higher or equal to that of sl2 and the categories of sl1 are a superset of those of sl2
- *domby* – sl1 dominates sl2, if the sensitivity of sl1 is lower or equal to that of sl2 and the categories of sl1 are a subset of those of sl2
- *eq* – sl1 equals sl2, if the sensitivities are equal and the set of categories are equal
- *incomp* – if sl1 and sl2 can't be compared, that is, neither is a subset of the other

To simulate the no-write-down behaviour of Bell-LaPadula, we could use the following rule:

```
mlsconstrain file write (l1 domby l2)
```

That means a process is not allowed to write to a file if it's security level is dominated by the security level of the file. To be correct, we must theoretically also constrain other permissions like **append**, because they also allow writing in a special meaning.

The **mlsvalidatetrans** statement is similar to its non-MLS variant. In contrast to that, the MLS-variant is more common. The differences are the respective keywords for the security levels, similar to the case of the **mlsconstrain** statement.

4.6 Multi-Category-Security

The Multi-Category-Security (MCS) implementation in SELinux can be seen as a replacement for MLS, as MLS only makes sense in the military sector, but not for a modern

operating system that is used on desktop systems. The very important difference between MLS and MCS is that MLS is still MAC, whereas MCS is DAC. And that is quite interesting from the users point-of-view. With MLS we have an SELinux enhancement that is MAC. With MCS we have an SELinux enhancement that is DAC. So the ordinary user gets the possibility to control his files with it.

For MLS the security context had to be modified, that is, the two security levels were added. MCS uses this security context with the difference, that the sensitivity levels are ignored and everything is labeled with the same sensitivity(s0). So MCS uses the MLS mechanisms, but the concept is not MLS.

To enable a subject to access an object, the user, who is the owner of this object(file), labels it with a category. The accessing process must be in this certain category to access the object. To have access to a category, the *high* range of the subject contains a list of allowed categories.

Since MCS is DAC and therewith granting permission lies within the discretion of the owner of a resource, MCS access control checks are made after the DAC checks and the TE rules are applied. Categories are commonly aliased so the ordinary user doesn't have to deal with cryptic categories like c0, c1 and so on.

The categories for a system user are assigned by defining a login for this user. To change the categories of a file and of an SELinux user, the `chcat` command is used.

```
1  # add categories to a file
2  chcat -- +Category1,+Category2 testfile.txt
3  # remove category from a file
4  chcat -- -Category2 testfile.txt
5  # add login for systemuser default
6  semanage login -a default
7  # give a user access to Category1
8  chcat -l -- +Category1 default
```

Listing 9: chcat command

4.7 Policy Files

When you need to write a new policy or look at the modules of the reference policy, there are 3 types of files for each module. The SElinux policy makes heavy use of the macro language m4. So it might be useful to read a bit about m4 when working more extensively with SELinux.

4.7.1 File Context Files (.fc)

Those files define default contexts for files that the module uses. If you use a command like `restorecon` or a distribution specific script that relabels the complete filesystem, this default context is applied to the files. Also, when you copy a file this security context will be applied to the copy.

```
1  /usr/sbin/aide          --      gen_context(system_u:object_r: ↷
       aide_exec_t,mls_systemhigh)
```

```
2
3 /var/lib/aide(/.*)              gen_context(system_u:object_r:aide_db_t ↷
      ,mls_systemhigh)
4
5 /var/log/aide(/.*)?             gen_context(system_u:object_r: ↷
      aide_log_t,mls_systemhigh)
6 /var/log/aide\.log      --      gen_context(system_u:object_r: ↷
      aide_log_t,mls_systemhigh)
```

Listing 10: aide.fc

Listing 10 shows the file context definitions for the Intrusion Detection Environment `AIDE`. Entries in .fc files consist of three columns.

1. The first column contains a regular expression that defines the file(s). This regular expression is anchored at both ends, that means it matches full pathnames, not only parts of it.

2. The second column can be used to limit the filetype. It can be left empty to not define the filetype any further. Or it can be in the form `-X`, where `X` is the filetype you can also see with `ls -l`.

3. The third column makes the security context. `gen_context` is a macro defined in the reference policy. Here you can see that the macro takes two arguments, the first one is the security context. The second one is optional and indicates the MLS/MCS range/category. You sometimes also find just the security context, but using the macro is the way to go for newer policies.

   Use <<none>> to explicitly not label the specified files. This can be used for files that have to be labeled at runtime, e.g. files in `/var/`.

4.7.2 Type Enforcement Files (.te)

These files make the heart of a module. They contain the Type Enforcement rules. When writing policy modules, you can use standard AV rules or m4 macros. If you look at listing 11, you notice that lines that contain a standard rule end with a `;`, where lines with m4 code don't.

```
1
2 policy_module(aide, 1.5.0)
3
4 ########################################
5 #
6 # Declarations
7 #
8
9 type aide_t;
10 type aide_exec_t;
11 application_domain(aide_t, aide_exec_t)
12
13 # log files
```

```
14 type aide_log_t;
15 logging_log_file(aide_log_t)
16
17 # aide database
18 type aide_db_t;
19 files_type(aide_db_t)
20
21 ########################################
22 #
23 # aide local policy
24 #
25
26 allow aide_t self:capability { dac_override fowner };
27
28 # database actions
29 manage_files_pattern(aide_t, aide_db_t, aide_db_t)
30
31 # logs
32 manage_files_pattern(aide_t, aide_log_t, aide_log_t)
33 logging_log_filetrans(aide_t, aide_log_t, file)
34
35 files_read_all_files(aide_t)
36
37 logging_send_audit_msgs(aide_t)
38
39 seutil_use_newrole_fds(aide_t)
40
41 userdom_use_user_terminals(aide_t)
```

Listing 11: aide.te

The first line of a .te file contains the `policy_module` macro. It defines the module name and version. The rest of the file make the rules discussed in Chapter 4.3.

4.7.3 Interface Files (.if)

In an interface file you can define macros in m4 language that can be used by other modules. A reason to use macros is to abstract a bunch of rules that is normally needed to accomplish a task within a module to a macro name. The advantage of using macros is that if you change your module, someone else who uses it doesn't need to take care of those changes.

```
1  ## <summary>Aide filesystem integrity checker</summary>
2
3  ########################################
4  ## <summary>
5  ##      Execute aide in the aide domain
6  ## </summary>
7  ## <param name="domain">
8  ##      <summary>
9  ##      Domain allowed access.
10 ##      </summary>
11 ## </param>
```

```
12 #
13 interface(`aide_domtrans',`
14         gen_require(`
15                 type aide_t, aide_exec_t;
16         ')
17
18         corecmd_search_bin($1)
19         domtrans_pattern($1, aide_exec_t, aide_t)
20 ')
21
22 ########################################
23 ## <summary>
24 ##      Execute aide programs in the AIDE domain.
25 ## </summary>
26 ## <param name="domain">
27 ##      <summary>
28 ##      Domain allowed access.
29 ##      </summary>
30 ## </param>
31 ## <param name="role">
32 ##      <summary>
33 ##      The role to allow the AIDE domain.
34 ##      </summary>
35 ## </param>
36 #
37 interface(`aide_run',`
38         gen_require(`
39                 type aide_t;
40         ')
41
42         aide_domtrans($1)
43         role $2 types aide_t;
44 ')
45
46 ########################################
47 ## <summary>
48 ##      All of the rules required to administrate
49 ##      an aide environment
50 ## </summary>
51 ## <param name="domain">
52 ##      <summary>
53 ##      Domain allowed access.
54 ##      </summary>
55 ## </param>
56 ## <rolecap/>
57 #
58 interface(`aide_admin',`
59         gen_require(`
60                 type aide_t, aide_db_t, aide_log_t;
61         ')
62
63         allow $1 aide_t:process { ptrace signal_perms };
64         ps_process_pattern($1, aide_t)
65
```

```
66          files_list_etc($1)
67          admin_pattern($1, aide_db_t)
68
69          logging_list_logs($1)
70          admin_pattern($1, aide_log_t)
71  ')
```

Listing 12: aide.if

4.8 Audit Log

When writing a policy you need to know what rules to implement. Although you should normally know what the rule should allow or not, you probably won't manage to get all necessary rules at once. To make your life easier, you will use tools like `audit2allow` that help you analyzing the logfiles and create corresponding rules. There is the `-w` option of `audit2allow` which helps at least filtering AVC rules out of the log jungle. Although those tools are quite helpful, they can also be dangerous since they only show you the generated rules. Since a rule could match on more than one program, you may want to inspect the situation a bit further and see if you need to define new domains to separate a process from others. So we should discuss the AVC log entries a bit, because this is needed for auditing.

```
avc:  denied  { write } for  pid=19774 comm="vim" name=".viminfo.tmp" ↷
     dev=hda1 ino=257658 scontext=root:staff_r:staff_t tcontext=root: ↷
    object_r:sysadm_home_dir_t tclass=file
```

Listing 13: auditing log entry

In listing 13 you see an AVC denial. This logfile entry consists of several parts:

- avc: – this shows that the entry is SELinux relevant, it has been caused by the AVC
- denied – an action has been denied, sometimes here may also be a "granted", which shows that an action has been explicitly allowed
- write – the type of access that was denied
- for pid=19774 – shows the PID of the process whose action was denied
- comm="vim" – the command "vim" tried to access an object
- name=".viminfo.tmp" – the command tried to access the object ".viminfo.tmp"
- dev=hda1 – the partition on which the object is stored
- ino=257658 – the inode number of the file
- scontext=root:staff_r:staff_t – the security context of the subject (source context)

- tcontext=root:object_r:sysadm_home_dir_t – the security context of the object (target context)
- tclass=file – the objects class

```
1 #============= staff_t ==============
2 allow staff_t sysadm_home_dir_t:file write;
```

Listing 14: TE rule created by audit2allow from the log entry in listing 13

5 Basic Commands

There are even more utilities available in the libselinux and policycoreutils packages than mentioned here, but this list should be enough to accomplish most tasks. If you need further information on a command you should read the corresponding manpage [13] or use the `--help` option of the command.

- **sestatus** – This command gives you information whether selinux is enabled or not, in which mode and what policy is loaded.
- **seinfo** – seinfo gives information about the currently loaded policy.
- **setenforce/getenforce** – getenforce tells you if the system is currently in enforcing or permissive mode. With setenforce you can enable or disable enforcing mode.
- **newrole** – With newrole you can change your your current security context. You will probably need this command to change your current role, e.g. `newrole -r sysadm_r`.
- **audit2allow** – This tool analyzes a given logfile for deny messages from the AVC and creates TE rules. Some important options are:
  - **-a** – uses the audit log as input (normally used)
  - **-d** – uses dmesg as input (sometimes needed when analyzing messages from the boot process, before auditd is loaded)
  - **-i logfile** – could be used on /var/log/messages for example to have similar results as -d
  - **-l** – you will always want to use this, only considers log messages after last policy reload (e.g. semodule -R)
  - **-R** – generates reference policy rules (uses macros if possible instead of plain rules)
  - **-m modulename** – specify a modulename (used to generate the policy_module statement at the beginning of a .te file)
- **semodule** – One of the most important tools when writing your own policy modules. Here are the most important options:

  - **-R** – reloads the policy
  - **-i module.pp** – install a new policy module
  - **-u module.pp** – upgrade an existing policy module
  - **-r module** – remove a module
  - **-l** – lists the currently loaded modules

- **semanage** – This utility enables you to manage many every-day tasks. It is used to create new SELinux users or to manage what linux users are allowed to login as a specify SELinux user. It is also used to manage network ports (e.g. allow apache to listen on port 81). With newer reference policies you can also set a per-domain permissive mode with semanage.

  It can do even more things. As always you can find out about this in the man-page. [13]

- **setsebool/getsebool** – Those tools are used to view or set the status of a SELinux boolean value.

- **chcon** – This command sets the security context of a file. It can either be used to set the complete security context or portions of it(user, type, ...).

- **restorecon** – This command restores the default security context for a file. It is often used to set security contexts when writing new policy modules.

- **common linux commands** – Most commands like `ls`, `ps` or `id` are normally patched on SELinux systems to show security contexts. They have the extra option **-Z**.

6 Usage Examples

Now that we have defined most of the important commands and terms let's have a look at how things are done on SELinux by some examples.

6.1 Adding a User

Adding a user on a classic linux system is quite an easy task. Using a program like `useradd` or even doing it manually doesn't require more than a minute. On an SELinux system there may be a bit more work to do. If you just want to add a standard user, you probably won't have to care about anything else. If you want the user to have special permissions, you will probably have to configure a new SELinux user and give it access to another role.

Suppose you want to add a new user called **newuser**. The user should have access to the roles `staff_r` and `user_r`(Note, that access to those two roles does normally not make sense, since they are nearly the same.).

```
1 root@SEStrict ~ # useradd -m newuser
2 root@SEStrict ~ # semanage user --roles 'user_r staff_r' --prefix user
    --add newuser_u
3 root@SEStrict ~ # semanage login --add --seuser newuser_u newuser
4 root@SEStrict ~ # passwd newuser
5   New UNIX password:
6   BAD PASSWORD: it is WAY too short
7   Retype new UNIX password:
8   passwd: password updated successfully
9 root@SEStrict ~ # chcon -R -u newuser_u /home/newuser/
```

Listing 15: adding a new user to your system

With `useradd` we add the user *newuser* to the linux system.

The `semanage user` command adds an SELinux user to the system and assigns the roles `user_r` and `staff_r` to this user.

The `semanage login` command connects the SELinux user *newuser_u* to the linux user *newuser*.

`passwd` sets the password for the user as usual.

Last but not least we change the security context for the user's homedirectory.

6.2 Using Booleans

```
1   root@SEStrict ~ # getsebool -a
2 ...
3 use_nfs_home_dirs --> off
4 use_samba_home_dirs --> off
5 user_direct_mouse --> off
6 user_dmesg --> off
7 user_ping --> off
8 user_rw_noexattrfile --> off
9 ...
10  root@SEStrict ~ # setsebool -P user_dmesg=on
11  root@SEStrict ~ # getsebool user_dmesg
12 user_dmesg --> on
```

Listing 16: listing booleans with getsebool

In Listing 16 we see the (shortened) output of `getsebool` and `setsebool`. Here we set the boolean `user_dmesg`. This boolean, if enabled, allows a common user to use the `dmesg` command. We also use the **-P** option to make the setting permanent, so it will survive a reboot of the system.

6.3 Inspecting the logfiles

The logfiles that are important for SELinux administrators may vary on different linux distributions. Normally you run the auditd [12] daemon on SELinux systems, which then logs to the file configured in `/etc/audit/auditd.conf`. On a gentoo system this is normally `/var/log/audit/audit.log`.

If auditd is not running, e.g. during the boot process before auditd gets started, or if you just don't use it, all the SELinux relevant log messages will be handled by the syslog daemon. On a gentoo system with syslog-ng this is normally `/var/log/messages`.

Whatever logfile you inspect, the most interesting entries will probably be the ones made by the AVC. If you would have to comb through the `audit.log` or syslog, you won't get much work done, since there will be much more entries than you need. Normally you only need to check the logfiles if something doesn't work as expected and you want to change the policy. To determine what didn't work, we will use the command `audit2allow`.

```
1   root@SEStrict ~ # semodule -R
2   root@SEStrict ~ # audit2allow -la
3  #============= user_t ==============
4  allow user_t proc_kmsg_t:file getattr;
5   root@SEStrict ~ # audit2allow -laR
6
7  require {
8          type user_t;
9  }
10 require {
11         type user_t;
12 }
13
14 #============= user_t ==============
15 kernel_getattr_message_if(user_t)
16   root@SEStrict ~ # audit2allow -lae
17 #============= user_t ==============
18 # audit(1230787789.236:41):
19 #  scontext="user_u:user_r:user_t" tcontext="system_u:object_r: ↷
       proc_kmsg_t"
20 #  class="file" perms="getattr"
21 #  comm="ls" exe="" path=""
22 #  message="type=AVC msg=audit(1230787789.236:41): avc:  denied  { ↷
       getattr } for
23 #   pid=4961 comm="ls" path="/proc/kmsg" dev=proc ino=4026531948
24 #   scontext=user_u:user_r:user_t tcontext=system_u:object_r: ↷
       proc_kmsg_t
25 #   tclass=file"
26 allow user_t proc_kmsg_t:file getattr;
```

Listing 17: listing AVC log entries with audit2allow

In Listing 17 we see different calls of the `audit2allow` command. Before using audit2allow we reload the policy with `semodule -R`. The first time we use audit2allow with the options **-l** and **-a**. **-a** tells audit2allow to use the auditlog as input. **-l** is to only consider entries since the last policy reload. We see a rule that, if used, would allow the `user_t` domain to get the attributes of `/proc/kmsg`.

The next use of audit2allow has the additional option **-R**. This option generates "reference policy style" output. That means it will search through interface files and see if it finds a macro that does exactly the same as an allow rule. It also generates the needed require commands. **(Note: there is a small flaw in some older versions**

which prints the require statement twice, sometimes even 2 slightly different require-blocks.)

To see why this rule was really generated we use the **-e** option in the third use. audit2allow prints the corresponding log message optically prepared. This can be very useful to see if you really want to use this rule or need to achieve your goal differently.

6.4 Writing a new Policy Module

Most likely you won't edit the base policy unless you're a maintainer of a linux distribution. But maybe you want the policy behave different. In this case you should probably rebuild your base policy without the part that is responsible for this certain behaviour and compile it as a module. At least that was one of the main reasons for inventing the reference policy.

If you need to write a completely new module, you will also want to make use of the `audit2allow` tool.

```
1  root@SEStrict ~/userkmsg # audit2allow -laRm userkmsg -o userkmsg.te
2  root@SEStrict ~/userkmsg # bzcat /usr/share/doc/selinux-base-policy
       -20081210/Makefile.example.bz2 > Makefile
3  root@SEStrict ~/userkmsg # ls
4  Makefile  userkmsg.te
5  root@SEStrict ~/userkmsg # #check userkmsg.te if it really does what
       you want
6  root@SEStrict ~/userkmsg # make
7  Compiling strict userkmsg module
8  /usr/bin/checkmodule:  loading policy configuration from tmp/userkmsg.
       tmp
9  /usr/bin/checkmodule:  policy configuration loaded
10 /usr/bin/checkmodule:  writing binary representation (version 8) to tmp
       /userkmsg.mod
11 Creating strict userkmsg.pp policy package
12 rm tmp/userkmsg.mod.fc tmp/userkmsg.mod
13 root@SEStrict ~/userkmsg # ls
14 Makefile  tmp  userkmsg.fc  userkmsg.if  userkmsg.pp  userkmsg.te
15 root@SEStrict ~/userkmsg # semodule -i userkmsg.pp
16 root@SEStrict ~/userkmsg # semodule -l|grep userkmsg
17 userkmsg        1.0
18 root@SEStrict ~/userkmsg #
```

Listing 18: creating a new policy module

After the .te file has been generated by audit2allow, be sure to check it manually if it really does what you want. An example Makefile is, at least on Gentoo Linux, installed together with the base policy and creates the .if and .fc files automatically.

References

[1] SELinux by Example, book by three authors that are quite involved in the development of SELinux – http://www.selinuxbyexample.com/

[2] Red Hat SELinux Guide – http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/selinux-guide/ (PDF: http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/pdf/rhel-selg-en.pdf)

[3] Gentoo SELinux Handbook – http://www.gentoo.org/proj/en/hardened/selinux/selinux-handbook.xml

[4] NSAs SELinux homepage, some good papers, with in-detail information about SELinux – http://www.nsa.gov/research/selinux/

[5] homepage of the FLASK – http://www.cs.utah.edu/flux/fluke/html/flask.html

[6] reference policy from tresys – http://oss.tresys.com/projects/refpolicy

[7] documentation of the reference policy, list of permissions, API documentation and much more – http://oss.tresys.com/projects/refpolicy/wiki/Documentation

[8] list of object classes and permissions – http://oss.tresys.com/projects/refpolicy/wiki/ObjectClassesPerms

[9] paper on the concepts of the reference policy – http://selinux-symposium.org/2006/papers/05-refpol.pdf

[10] statement of assurance regarding type enforcement from "Secure Computing" – http://www.securecomputing.com/pdf/Statement_of_Assurance.pdf

[11] LSM design – http://www.usenix.org/event/sec02/wright.html

[12] Homepage of the auditd daemon – http://people.redhat.com/sgrubb/audit/

[13] online manpages – http://linux.die.net/man/

[14] Integrating Flexible Support for Security Policies into the Linux Operating System – http://www.nsa.gov/research/_files/selinux/papers/freenix01-abs.shtml

[15] A Brief Introduction to Multi-Category Security (MCS) – http://james-morris.livejournal.com/5583.html

[16] Getting Started with Multi-Category Security (MCS) – http://james-morris.livejournal.com/8228.html