**Adventures in Creating a Custom SELinux Policy**

NCSU Skyward Federal Senior Design Team Spring 2020

Author: Spencer Yoder

Editor: Daniel Mills

**What is an SELinux Policy?**

An [SELinux](https://www.redhat.com/en/topics/linux/what-is-selinux) policy, at its core, is a set of rules which define which processes (domains) can do which actions (permissions) to which objects (types). These concepts will be expanded on in a second. Every process and file in an SELinux system is assigned a [security context](https://fedoraproject.org/wiki/Security_context), which by itself is just a label. Policies define how contexts can interact in certain ways. By default, all actions which aren’t defined in the policy are not allowed to occur on the system.

**Some Definitions:**

Type Enforcement - This is what SELinux does. Type enforcement refers to fine-grained access control where processes of certain types can only interact with other types in certain ways.

Targeted Policy - In SELinux, this is the default policy. The targeted policy specifies that all actions not defined in the policy are denied *except* for actions taken by processes running with the domain unconfined\_t. This domain is not affected by SELinux. This policy is activated by running

**sudo setenforce** **1**

if SELinux is set in targeted mode in /etc/sysconfig/selinux.

Permissive Policy - The permissive policy enforces no selinux rules. *However*, it still logs every action which would be denied. This is a good policy to activate when debugging your policy and is activated by running

**sudo setenforce 0**

MLS Policy - This policy uses the level field of security contexts. If MLS is activated, it is used in tandem with the other active policy to further constrain data. There’s a little bit more about MLS [here](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/mls). We didn’t use MLS for our custom policy.

Security Context - A label assigned to a Linux object (user, process, file, etc.) which is formatted as follows: user:role:type:level.

User - Has no correlation to a Linux user. SELinux users are simply a way to constrict which roles an object can be assigned. Certain users can only be assigned certain roles.

Role - Signifies which types an object can be assigned.

Type - For the targeted policy, which we used, this is the most important part of the security context. SELinux rules regarding access only apply to types. OS transactions occur between a source and a target (like a process and a file). When a type is applied to a source, it is called a domain. When it’s applied to a target, it’s just called a type.

Class - Every SELinux type has a class. A class just lets the policy know what kinds of things can be done to that type. For example, types of the ‘file’ class have actions read, write, execute, etc. associated with them.

Domain - The type of a running process. Policy rules identify which domains can interact with which types and how. Note that if a process wants to do something to another process, the second process’s domain is its type. A type is a domain only if the object with said type is the one performing the action.

Level - The last field of the security context. Only has meaning in the MLS policy. A level is comprised of a sensitivity range and a category range. There’s more about levels [here](https://selinuxproject.org/page/NB_MLS).

AVC (Access Vector Cache) Rules - Or just, “AV Rules,” but audit logs refer to AVC denials, so I’ll include the cache part. These are the rules which determine what happens when a particular domain tries to interact with a particular type in a certain way. When you create a policy, these are usually the meat of said policy. More about them [here](https://selinuxproject.org/page/AVCRules).

Audit Logs - In SELinux, every transaction is logged. Audit logs are stored in /var/log/audit/audit.log but can be searched with ausearch.

Ausearch - [This command](https://linux.die.net/man/8/ausearch) lets you search through the audit logs for particular elements. We used this to debug our security policy. A strategy which worked really well was the sequence of commands:

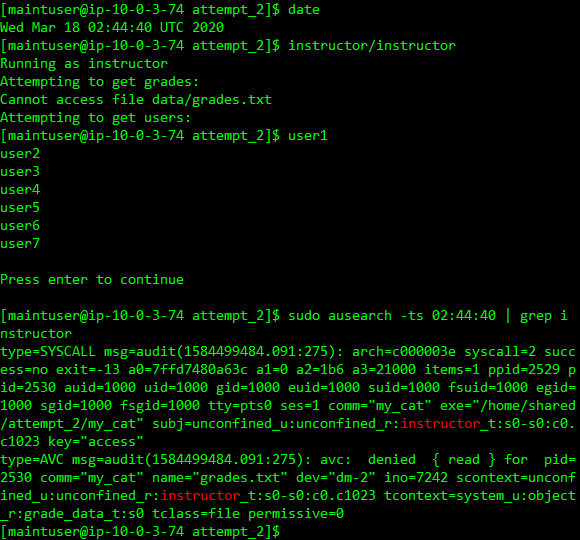
**date**

**<command I want to debug>**

**sudo ausearch -i -ts <time specified from date> | grep <command name>**

where date gives the current system date and time, the command is some command we want to debug, the -i flag interprets certain parts of the log to convert them to ascii text, and the -ts flag signals ausearch to only search after the time we enter after it.

Here’s an example of what this looks like:



Note that I forgot to use the -i flag here.

**Necessary Tools**

Before we can create our own security policy, we need the tools. Our group ran

**sudo yum install** [**libselinux-utils**](https://www.mankier.com/package/libselinux-utils)

I think we installed this at a previous step, but a really helpful associated tool is:

* [setenforce](https://linux.die.net/man/8/setenforce)
  + Changes SELinux between enforcing (targeted) and permissive mode

**sudo yum install** [**setools-console**](https://www.mankier.com/package/setools-console)

Pertinent tools:

* [seinfo](https://linux.die.net/man/1/seinfo)
  + Very helpful for finding out existing types, rules, etc. in your current policy.
* [sesearch](https://linux.die.net/man/1/sesearch)
  + Helpful for finding AV rules which apply to certain types/domains
* [sestatus](https://linux.die.net/man/8/sestatus)
  + Shows broad details about the policy SELinux is currently enforcing

**sudo yum install** [**policycoreutils-python**](https://centos.pkgs.org/7/centos-x86_64/policycoreutils-python-2.5-33.el7.x86_64.rpm.html)

Pertinent tools:

* [semanage](https://linux.die.net/man/8/semanage)
  + Has the ability to add certain (not all) policy rules from the command line
* [semodule](http://man7.org/linux/man-pages/man8/semodule.8.html)
  + The tools which adds modules to the current policy and can be used to rebuild the current policy
* [sepolicy](http://man7.org/linux/man-pages/man8/sepolicy.8.html)
  + sepolicy generate is what helps us create the starter files for our new policy.

We already had [coreutils](https://www.gnu.org/software/coreutils/) installed

Which contains

* [chcon](https://linux.die.net/man/1/chcon)
  + Which allows you to change the context of files much like chmod (if the policy permits it)

Before continuing, it’s important to make sure you have all these commands. If you’re missing any, you can google them to find out how to install them.

[**sepolicy generate**](http://man7.org/linux/man-pages/man8/sepolicy-generate.8.html)

When you want to create a new policy, this is the first command you will run. Here’s what this will and won’t provide to you and how to run it.

Our team created a test policy, just meant to help us figure this stuff out. We had a very simple system: a program called my\_cat, written in C, which prints out the contents of a file to the terminal, two text files, one with grades, and one with usernames, and the following requirements:

* processes running as coordinator\_t should be able to access files with type grade\_data\_t and files with type user\_data\_t
* processes running as instructor\_t should only be able to access user\_data\_t
* processes running as student\_t should only be able to access grade\_data\_t

What we ended up doing was running sepolicy generate three times, one on an executable file for each domain. I’m getting a little ahead of myself though.

Here’s how to run sepolicy generate:

**sepolicy generate --application <path to binary>**

Ideally, your working directory should be an empty one, set aside for your policy files, sepolicy generate creates quite a few:

* application\_name.te
  + A type enforcement file. All of our changes ended up being in these kinds of files. This is where you write the rules for your new policy
* application\_name.if
  + An interface file. As near as I can tell, this file tells selinux how it interacts with certain types (like opening a new type of files)
* application\_name.fc
  + A ‘file context’ file. Handy in theory, it should assign new contexts to certain files, however we never got this to work as it should and ended up assigning file contexts with chcon
* application\_name.selinux.spec
  + This file seems to help with installing the new policy
* application\_name.sh
  + This is the file you run when you’re ready to compile and add your new module. Every time it’s run, it rebuilds the policy.

At the start, these files do very little. They create new types, application\_name\_exec\_t and application\_name\_t. Building the policy changes the context of your application binary file to application\_name\_exec\_t. The rest of the rules help give the file basic permissions (like the ability to read its own contents). Remember that selinux disallows all behavior not specified in the policy. Adding new types and rules will be up to you.

**A Series of Notes**

Depending on what you want to do, --application might not be the best option. We plan to create a policy for the postgres server, which right now is set up to run as a [daemon](https://en.wikipedia.org/wiki/Daemon_(computing)). In that case, we would use --init.

Remember that sepolicy generate makes template files. You will not be restricted based on the option you choose, but you may be denied certain basic accesses if you pick the wrong type. In that case, you would have to add them by hand.

Why did we create three new applications with three new policy modules? Initially, we just had the one policy module for one application. However, we found out about [context inheritance and context transitions](https://wiki.gentoo.org/wiki/SELinux/Tutorials/How_does_a_process_get_into_a_certain_context) and decided that the best way to run one application from three different contexts would be to start each new application from a different context and then run our original application (my\_cat) so that it would inherit the context of these separate processes. The easiest way to achieve this was to generate three new policy modules, one for each application, to guarantee that a lot of the messy stuff was handled by these template files.

[**PolicyLanguage**](https://selinuxproject.org/page/PolicyLanguage)

Since we’ve got a policy created, we can add our rules. On the PolicyLanguage section of the SELinux wiki, the only useful pages seem to be under these [definition links](https://selinuxproject.org/page/PolicyLanguage#Kernel_Policy_Language_Definition_Links).

**General syntax:**

This syntax refers to .te files. To figure out what should go in the other files, you will need to do some googling.

As near as I can tell, a .te file can have the following structures

<type> <name>;

<type> <name> <modifier type> <mod>;

<type> <name> <modifier type> {<mod 1> <mod 2>};

#^^ These are declarations. This is a comment. Declarations can:

#define new SELinux context objects:

role coordinator\_r;

type coordinator\_t;

#modify existing current objects:

role coordinator\_r types {coordinator\_t unconfined\_t};

#or both at one time:

user coordinator\_r roles coordinator\_r;

<rule> <domain> <type>:<class> <action>;

<rule> <domain> <type>:<class> {<action 1> <action 2>};

#^^ These are policy rules. We only ever used two of these

#in our test policies:

allow coordinator\_t user\_data\_t:file {read write open getattr};

#^^ This is an allow rule. Processes with domain coordinator\_t

#can read, write, open, and get attributes of files with type

#user\_data\_t

type\_transition unconfined\_t coordinator\_exec\_t:process coordinator\_t;

#^^Imagine this line is in 12-point font

#This is a type transition rule. When processes with domain

#unconfined\_t execute files with type coordinator\_exec\_t, they

#transition to domain coordinator\_t

require {

<type> <name>;

}

#^^Declarations inside a require block aren’t actually created

#in this policy. Pre-existing objects go here. This allows you

#to reference these objects without creating duplicates.

#Example:

require {

role object\_r;

type unconfined\_t;

}

#object\_r and unconfined\_t already exist in the policy

macro\_name(<parameter>)

#^^TE policy language allows the usage of macros.

#Anything structured like this with parentheses and no semicolon

#is a macro and expands into multiple lines. Some macros are

#[here](https://selinuxproject.org/page/RefpolicyWriteModule) but I’ve not been able to find a comprehensive list.

#We used the macro files\_type to assign our new types to the

#file class:

files\_type(grade\_data\_t)

Here is what our .te files for our new policies looked like: [coordinator.te](https://drive.google.com/file/d/1k4ZU7SxneCLf7PdEoEkr7gG8iSJcQqCx/view), [instructor.te](https://drive.google.com/file/d/1YOZrpF8p-Lt6Du1qYkcX2zf2tQls4gmM/view), [student.te](https://drive.google.com/file/d/1fki9NoW_ydNPEyS0_s1g58KgS4n17mOI/view). This is about all I can share about writing the policy, which, in my opinion, is the hardest part. Rely heavily on the policy language wiki and whenever you get an SELinux denial, google it to find out what your policy is missing.

**Developing Your Policy**

We found that developing this policy according to the following strategy was very helpful:

1. Make a small number of changes to the policy files.
   1. An example would be to add a new type:  
      **type grade\_data\_t;**
2. Compile the policy with  
   **sudo sh policy\_name.sh**
   1. If the policy compiles and builds successfully, proceed to step 4
   2. If not, proceed to step 3
3. Fix your compilation/build errors and then return to step 2
   1. The compilation error messages for these policies are usually straightforward. The only confusions we encountered were:
      1. Compilation errors for lines in macros. In this case, both the line number for the expanded policy and the line number for the unexpanded policy are displayed.
      2. For some reason, the policy would not compile unless user declarations were at the very end of the file. If you declare new selinux user types, make sure to do this, the compiler will not tell you what you did wrong.
   2. You may need to hunt for build errors. After compiling, the script prints out a pretty lengthy series of messages. It will mark build errors, though.
      1. The most common build error we got was for duplicate users/types.
4. Look for evidence that your changes have applied
   1. If you’ve built the policy for the first time, you can run  
      **ls -Z <path to executable>**  
      The -Z option shows the security context. If your executable was modified by the policy, it should have the type executable\_name\_exec\_t.
   2. One of the things we did was to add waiting for user input in the my\_cat executable. This way, we could launch another terminal, run our program, and run  
      **ps -uZ**   
      on the first terminal to see the context for all running processes.
   3. The best evidence that changes have applied is your program not being able to run. If you run  
      **sudo setenforce 0**  
      and your program can run, SELinux is preventing your program from running. Run  
      **sudo setenforce 1**  
      to re-enable SELinux and continue to step 5.
5. Debug your policy
   1. The most common type of error we encountered was SELinux not allowing us to do things after building the policy
      1. To find out what’s being denied, you can run the sequence of commands laid out in the ausearch section of the definitions above.
      2. If you can’t find the problem, or grep doesn’t print anything, you can remove the pipe to grep and manually look through the [audit log entries](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/sec-understanding_audit_log_files), there shouldn’t be too many to look through.
      3. If you still can’t find the problem, you can run  
         [**sudo semodule -DB**](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security-enhanced_linux/sect-security-enhanced_linux-fixing_problems-possible_causes_of_silent_denials)  
         to allow every single audit to be printed to the log. By default, [some messages do not get logged](https://selinuxproject.org/page/AVCRules#dontaudit). After you’re done debugging, make sure to run  
         **sudo semodule -B**  
         to allow the policy to ignore messages again. Logging every single message can affect the performance of your system.
      4. The prior three steps helped us debug all of our problems. If you’re looking through the whole audit log with --dontaudit disabled and still can’t locate the issue, this guide can no longer help you, good luck.
   2. The second most common type of error was our program being allowed to do things it shouldn't
      1. The first thing you should check is the program’s behavior when the line  
         **permissive policy\_name\_t;**  
         is commented out in your program. This line in a .te file runs a permissive policy only for this domain.
      2. If your program is still doing things it shouldn’t be able to, run it in another terminal and then run  
         **ps -uZ**  
         to check its security context. Programs running as unconfined\_t were the prime cause of these kinds of error for our group. We made our my\_cat executable wait for user input in order to be able to check its running context before it terminated.
      3. Remember that, for new domains (like the one defined in your new policy), SELinux denies every operation. As mentioned previously, the template policy generated gives this new context some very basic permissions, like the ability to run as a context at all. If your program is accessing things it shouldn’t, it’s either:
         1. Running in a different domain
         2. Running as permissive
   3. In general, finding comprehensive documentation for SELinux policies was very difficult, but finding fixes by googling specific errors was much easier. If you’ve managed to debug your specific change, start back again from step 1 until your policy is complete.

**Resources**

If you ever need to remove your policy, you can do so with

**sudo semodule -r <policy\_name>**

* Our C files and TE files for our practice policy:
  + [coordinator.c](https://drive.google.com/file/d/1unZOuz20zTezfIbglp1trMf2m8DqXA_3/view)
  + [instructor.c](https://drive.google.com/file/d/1Uv0IaScgB5yYj99azSJTiv3SI0iOo3Ha/view)
  + [student.c](https://drive.google.com/file/d/1mZZN8_De71scHr__LWNf63vl5deNCjjV/view)
  + [my\_cat.c](https://drive.google.com/file/d/1kIsk_GuMFPqQSMQsAB8Ncm243ha0fFph/view)
  + [coordinator.te](https://drive.google.com/file/d/1k4ZU7SxneCLf7PdEoEkr7gG8iSJcQqCx/view)
  + [instructor.te](https://drive.google.com/file/d/1YOZrpF8p-Lt6Du1qYkcX2zf2tQls4gmM/view)
  + [student.te](https://drive.google.com/file/d/1fki9NoW_ydNPEyS0_s1g58KgS4n17mOI/view)
* A video showing how to build a simple policy
  + [example\_policy\_creation\_debug.mp4](https://drive.google.com/file/d/1tqFlnilLuthafD7JjkTGHSZfJfoFcqXX/view)