**Adventures in Writing an SELinux Security Policy Module for a RESTful service Using SEPostgres**

NCSU Skyward Federal Senior Design Team Spring 2020

Authors: Spencer Yoder, Jonathan Balliet

**Introduction**

The previous instruction docs for this project were written after the fact. Since I figure that this may be an even more complicated process than installing SEPostgres and learning how to write policy modules, I’ve opted to write down things as I go. This will sacrifice document structure for an increase in detail (hopefully).

**Useful Prior Info**

CourseManager

* [Requirements](https://docs.google.com/document/d/19GO2d7NewHKXSrCZtRAPQGpJqLTy6ywMsfniPWZ7Wmg/edit?usp=sharing)
* [Database Schema](https://docs.google.com/document/d/10xLrlY2cwH_9MHbjFbGnmopXPkt1WgSHtUXCWo14onA/edit?usp=sharing)
* [Security overview and initial design](https://docs.google.com/document/d/1u-1UdOm22C6px2Kf4NItKp4sWjDPVgROFHHWHxeB7NY/edit?usp=sharing)

[Installing SEPostgres](https://docs.google.com/document/d/1feUvzFwwyJDRhkZPnw_jfaSt4-X368f885zLaq-19Ug/edit?usp=sharing)

[Creating an SELinux Security Policy Module](https://docs.google.com/document/d/1ZuNzi3eEbFre7Sy_B9EmK9DYrUUbqH7YgKRZDWnFz98/edit?usp=sharing)

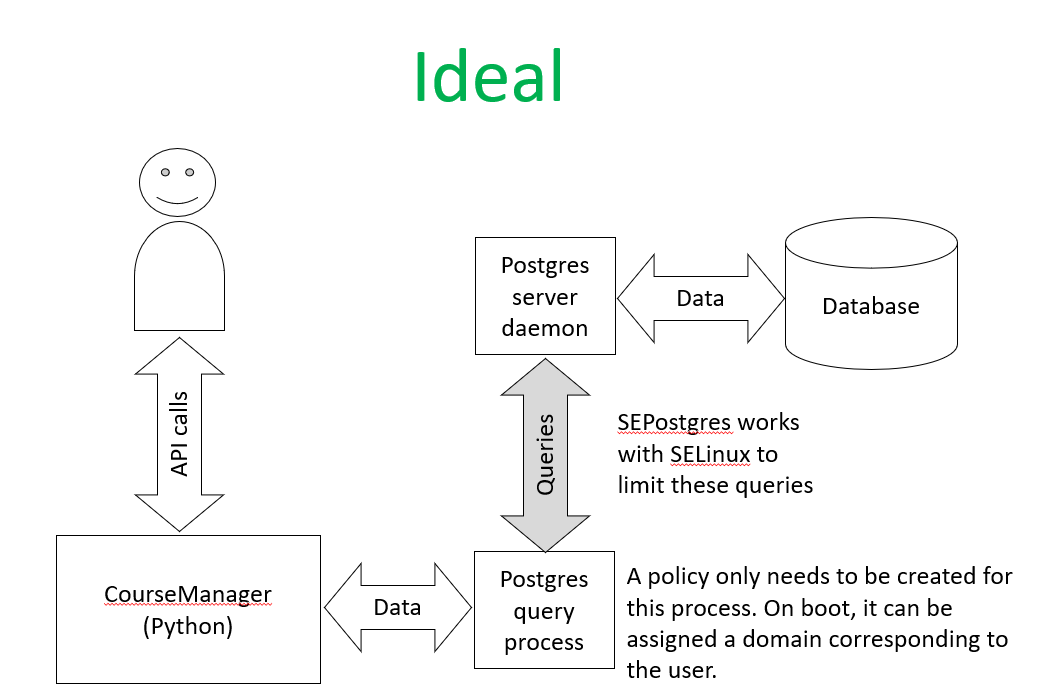
**Overview**

By the time this document is finished, the following should be true:

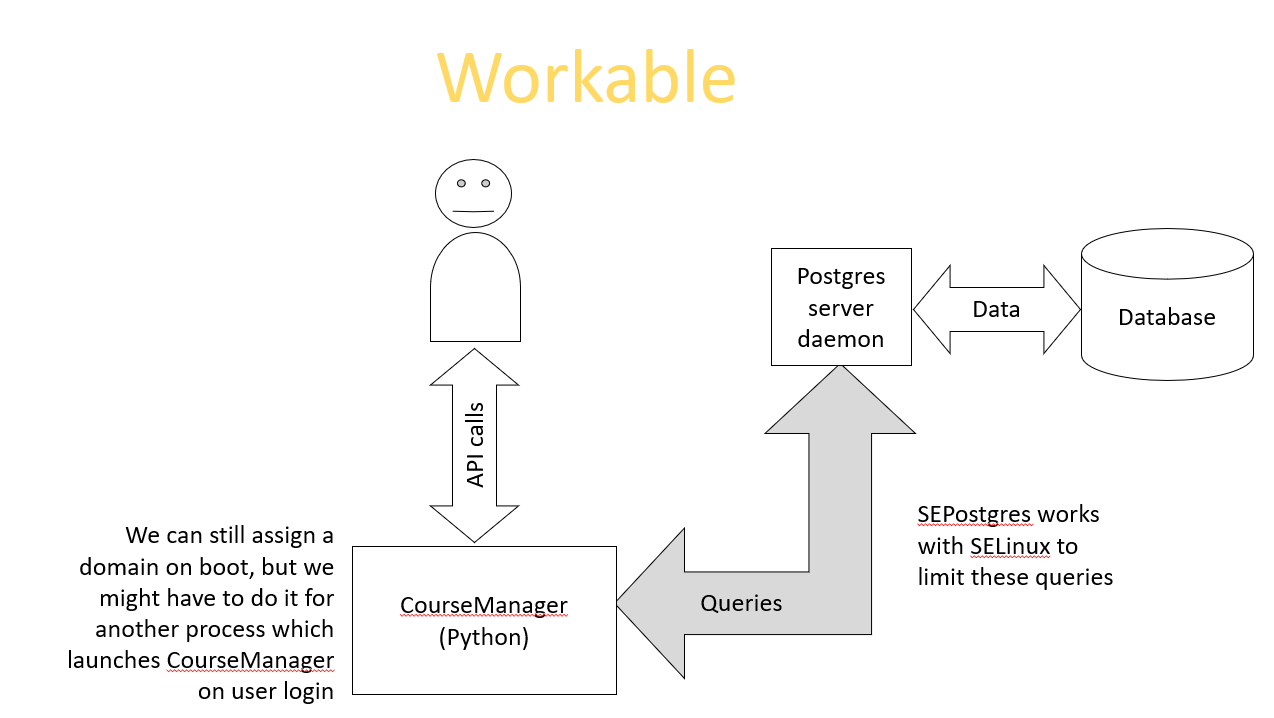
* CourseManager is installed on one of our CentOS 7 EC2 instances
* CourseManager is interfacing with the Postgres database
* CourseManager has successfully populated the Postgres database
* The processes CourseManager runs to interface with Postgres are known
* A mechanism has been put in place to allow the CourseManager service to launch with a user-appropriate context.
* An SELinux policy module has been written, compiled, and added which adheres to the rules in the CourseManager security document

The goal at the start of this phase of the project will be research. There are a few unknowns which need to be addressed before a policy can even be written:

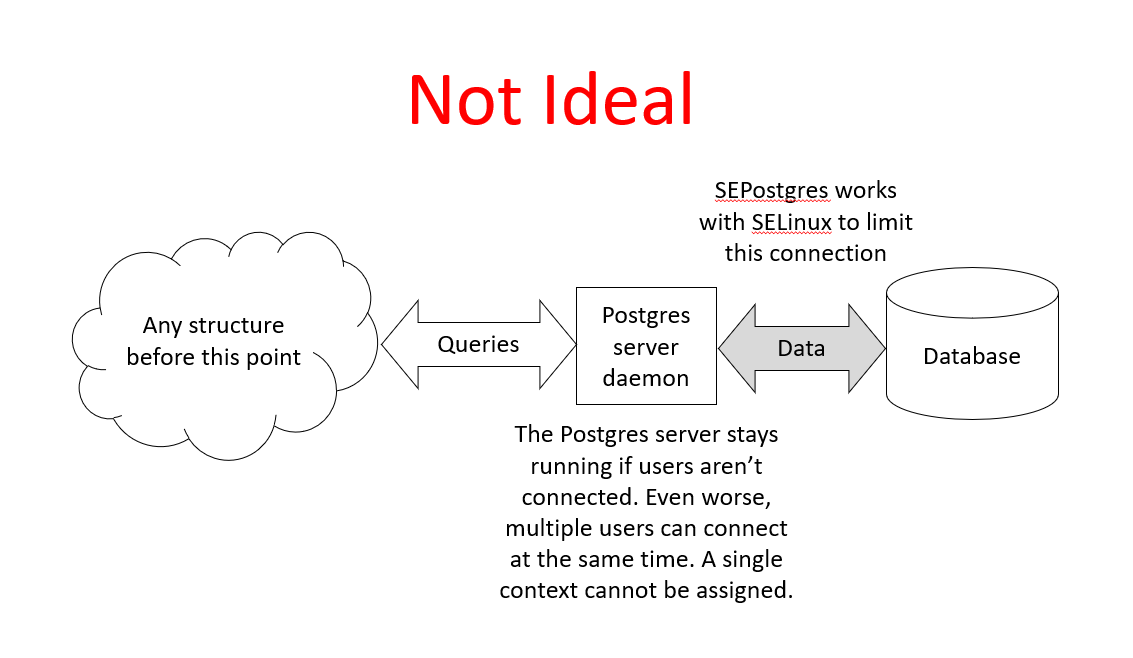
* How does our Python service talk to Postgres?
  + SELinux performs type enforcement on processes. Ideally, Python will launch a one-off process to handle database accesses. In this case, there will need to only be a set of rules for that process which just handle transition on startup.
* Which domain does SEPostgres check when handling database accesses?
  + The worry here is that even if Python starts a one-off process to access the database, SELinux might only perform checks on the postgres server. Even if the Python interface is properly secured, it will just get its data secondhand by talking to the postgres server.



*Figure 1*



*Figure 2*

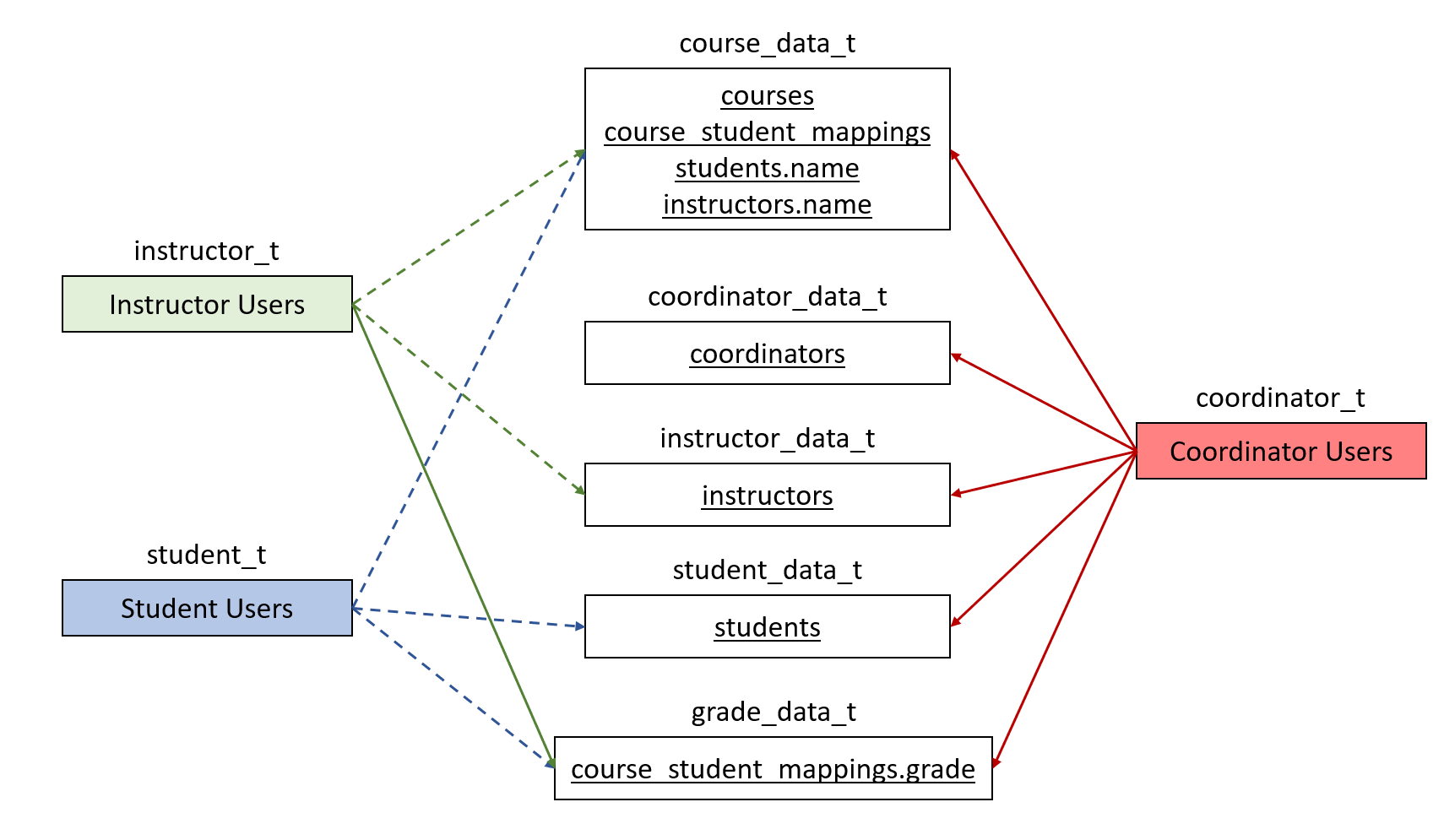


*Figure 3*

**Updated Security Policy**

When we were writing a sample security policy, we found out that the way we were thinking of SELinux was incorrect. We had planned out a modification of MLS, which, it turns out, is a small part of SELinux, the main part being type enforcement. There’s more on this in the security policy document.

Because of this, we have to update the policy plan from the security document to something like the following:



*Figure 4*

Where underlined items are database objects (students is a table, students.name is a column), dashed lines represent read permissions, and solid lines represent read/write permissions

**Installing CourseManager on our CentOS7 EC2 machines**

Jonathan, Daniel, and I joined forces to try and install CourseManager on the CentOS7 EC2 machines. We managed to learn a lot about SEPostgres and have new predictions about the shape of the system.

First, I followed Jonathan’s instructions [here](https://docs.google.com/document/d/1lLO9lk4K2TSZLCL19D1na8LnpR4jSDgo8XCYRtLFxH0/edit?usp=sharing).

I got an error when running one of the pip commands. This was because my volume was full. I doubt you, reader, will have this error, but I fixed it simply by deleting the Postgres source code I had downloaded earlier to free up some space.

All three of us got stuck on one error:

**sqlalchemy.exc.OperationalError: (psycopg2.OperationalError) FATAL: SELinux: unable to get peer label: Protocol not available**

When trying to run Jonathan’s script to populate the database.

We managed to find someone with the same error at [this website](http://www.2done.org/44.html). (Note that the website is in Japanese and needs to be translated). The error was caused because our Python script connects to the database as if it were a web client. SELinux tries to get the security context for this connection and is not able to since none is assigned.

The website suggests using **yum -y install netlabel\_tools** to use tools for configuring [labeled IPSec](https://tools.ietf.org/html/draft-ietf-ipsecme-labeled-ipsec-01). We installed the package and used

**sudo netlabelctl unlbl add default address:127.0.0.1 label:unconfined\_u:unconfined\_r:unconfined\_t:s0**

(all on one line)

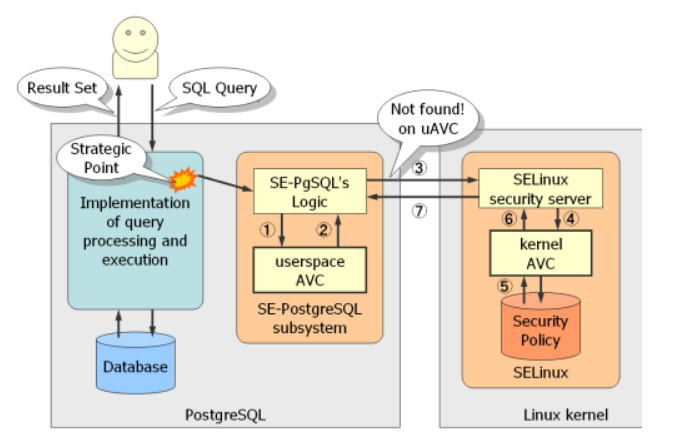
to assign the default label unconfined\_u:unconfined\_r:unconfined\_t to connections from localhost. Note that this label was chosen as a placeholder for our final label, which will correspond to the type of the calling user. After labeling this IP connection, we got the following error:

**sqlalchemy.exc.ProgrammingError: (psycopg2.errors.InsufficientPrivilege) SELinux: security policy violation**

Which is great, it signifies that the connection went through, SEPostgres just didn’t allow the operations the script was trying to perform.

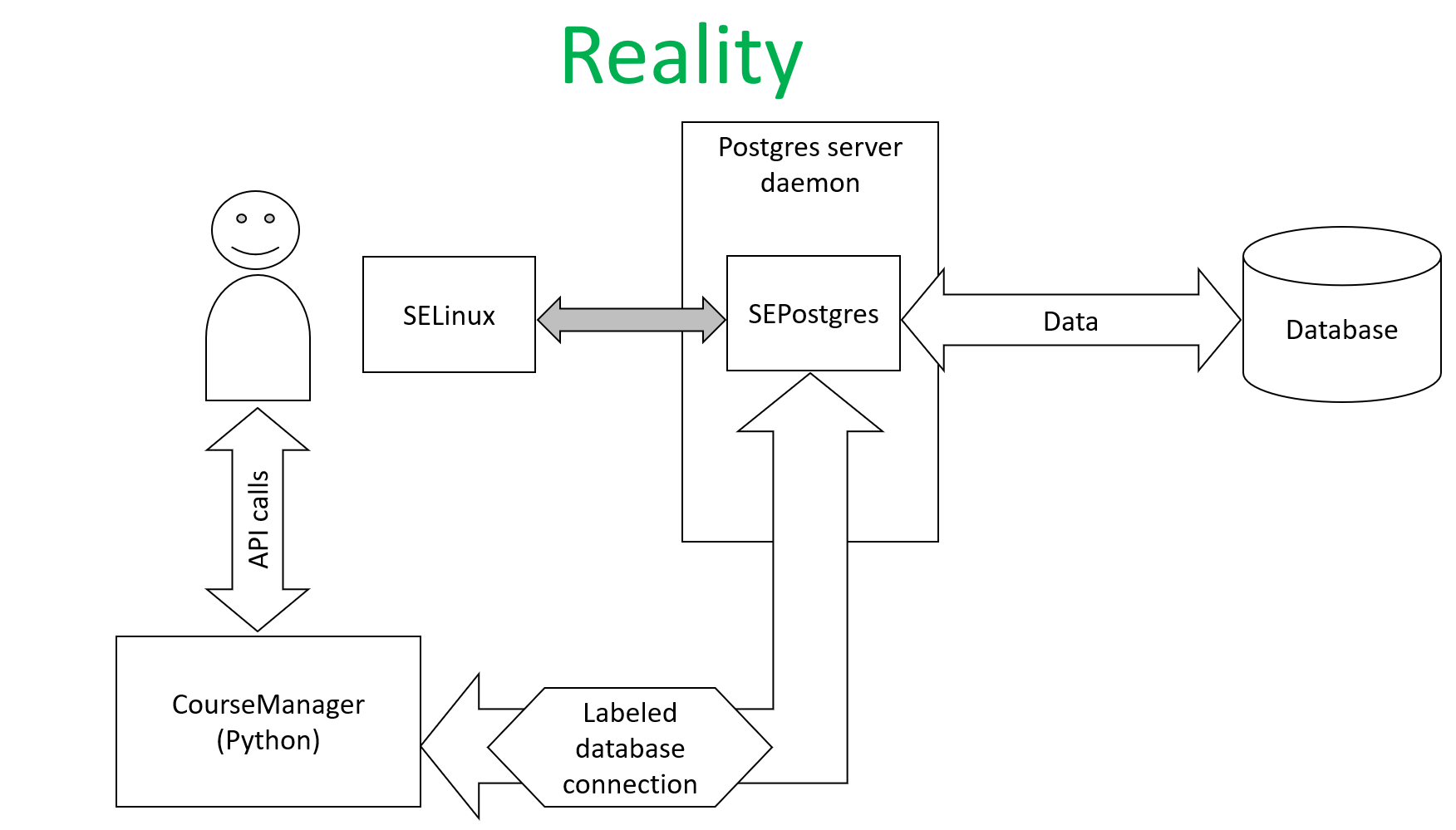
Through this process we learned a lot about how SEPostgres works. The first thing we found out was that SEPosgres might not enforce things through SELinux, it uses system calls to talk to SELinux. There’s some support for this theory in the fact that SELinux denials were not getting logged to the audit logs. We could only find our deinals by running [journalctl](https://manpages.debian.org/stretch/systemd/journalctl.1.en.html), which parses through systemd logs, implying that logging is done by SEPosgres, not SELinux.

[This page](https://wiki.postgresql.org/wiki/SEPostgreSQL_Architecture) of the SEPostgres documentation contains this diagram:



*Figure 5*

And states that when a query is sent by a user, SEPostgres checks its own [access vector cache](https://selinuxproject.org/page/NB_CoreComponents) to see if the query is allowed, if not, it sends a request to SELinux to do the same thing. Security-wise, this presents a problem because at least part of the access control happens in userspace. For this project, this could be good news.



*Figure 6*

The important thing about this view of the system is that SEPostgres sends SELinux attempted accesses and SELinux informs SEPosgres about whether those accesses are allowed. The former predicted models for these interactions assumed that a context would have to be assigned to some running process. This model means that all the policy needs is a list of access rules, with no instructions for context-switching.

A decrease in the amount of policy that needs to be written is great; however, this new model requires us to label incoming connections based on the type of user. We’re currently looking for a way to do that via the Docker, since we intend to spin up a new container running CourseManager for every new user

**Writing a Policy for CourseManager**

So, to begin writing a policy for CourseManager, we started with the decision to give a connection labeled as coordinator\_t:

**sudo netlabelctl unlbl add default address:127.0.0.1 label:course\_manager\_u:course\_manager\_r:coordinator\_t:s0**

permission to run the generate\_db.py script in the /cops\_platform/services/course\_manager/tests/db directory in our database. Before we could do this, it was necessary to revisit our method for creating SELinux policy modules.

The former method described in the SELinux policy module doc, can be used to create a policy for an exectable process and creates templates with the default permissions for allowing an executable to be run, running with a context transition, etc.

Since our current policy only needs to specify permissions for the types in our database, we don’t need all the extra rules. By following the instructions [here](https://relativkreativ.at/articles/how-to-compile-a-selinux-policy-package), we began to create a policy module just by writing a .te file. However, we got the error:

**'Building a policy module, but no module specification found.’**

when compiling with checkmodule. So we switched to using the command specified in [this bug report](https://bugzilla.redhat.com/show_bug.cgi?id=1367748) to compile our .te file to a .pp file. To shorten the command, we pasted it into a bash script in our policy directory called compile\_alias.sh so that:

**source compile\_alias.sh**

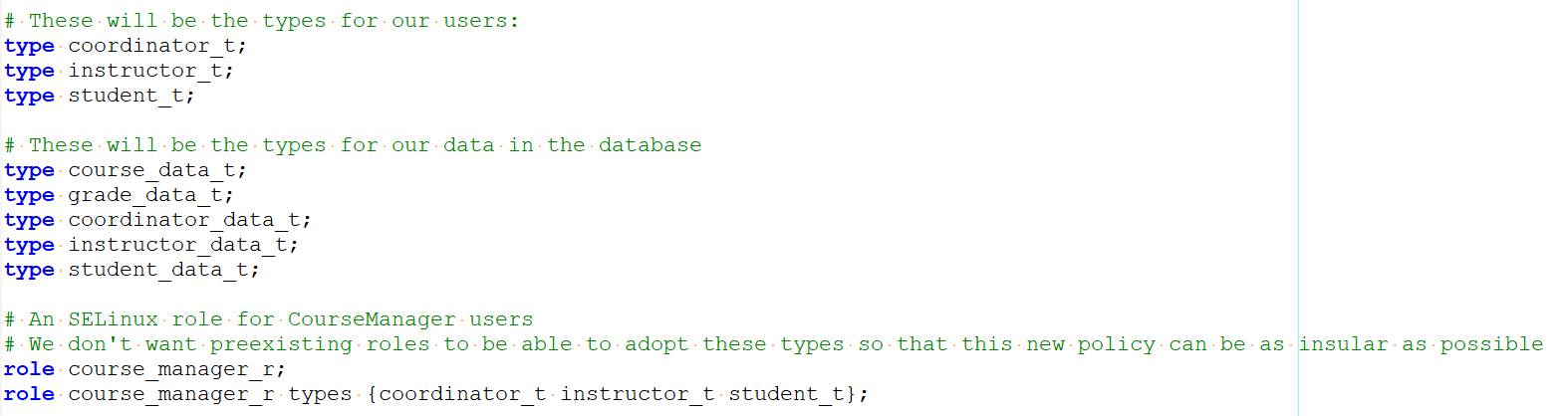
can be run and from there on,

**secompile**

could be used to compile course\_manager.te to course\_manager.te. Following that,

**sebuild**

can be run to build the module into our existing policy.



The first lines we wrote into our policy were those designating the new types. They correspond to the types detailed in Figure 4. After that we labeled the database. By running:

**psql -d johndb -U john -W -h 127.0.0.1**

We can use the database in the same way that python would. However, without any access vector rules written for coordinator\_t, we needed to run

**sudo setenforce 0**

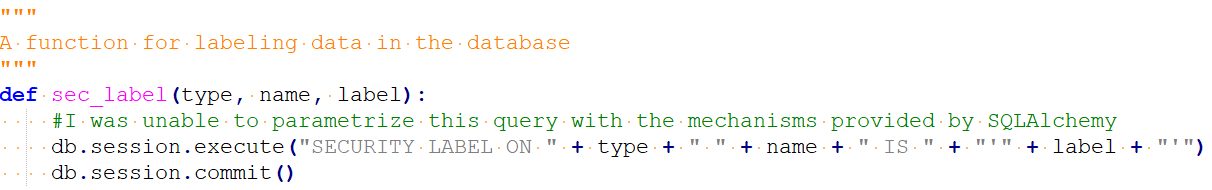
to change SELinux to permissive mode.

**security label on <table, column, etc.> <name> is ‘<label>’;**

is the Postgres command which labels database objects. Security labels can be viewed by running

**SELECT \* FROM pg\_seclabel;**

We ended up automating this in our generate\_db.py program.



Initially there were issues with the python script not labeling the columns/tables in the database. This was because the statement *db.engine.execute* was being used with the SQLAlchemy library, along with no commit statement, after executing these SQL instructions to the database. Changing this code to *db.session.execute*, followed by *db.session.commit,* resulted in successfully labeling the database.

**Properly labeling CourseManager communication**

We ended up creating a policy for CourseManager, but it required manually labeling localhost as the user we wanted to log in as. So, we switched gears and used our container runtime application to label the IPs for each container based on the conencted user. This can be done because the container runtime knows the user types at startup.

Initially, we attempted to label the containers themselves with the docker **--security-**

**opt** argument. This would not work on our machines, we think because of our version of Docker or SELinux.

So, we settled for labeling each container’s IP address. Here’s the control flow for how we do that:

<https://wiki.postgresql.org/wiki/SEPostgreSQL_References#common_database_class>

<https://wiki.postgresql.org/wiki/SEPostgreSQL_References>

<https://wiki.postgresql.org/wiki/SEPostgreSQL_Administration#Labeled_networks>

<https://www.slideshare.net/kaigai/label-based-mandatory-access-control-on-postgresql>

denied ingress for coordinator\_t to netif\_t, class netif

denied recv for unconfined\_t from coordinator\_t, class peer

denied recv for unconfined\_service\_t from coordinator\_t, class peer

denied recvfrom for coordinator\_t for node\_t, class node

TODO: Document attempts to pass in SELinux label to Docker and discuss errors we encountered and troubleshooting/debugging attempts.

In order to override the default labeling scheme of Docker Containers, we attempted to use the --security\_opt flag to pass in a SELinux label when using the “docker run” command. A precondition is that docker must be installed and working correctly with Course Manager flask application according to the steps detailed in the following set of instructions:

<https://docs.google.com/document/d/18KWS1BBJ4EwFL4c3Fh_LX9tBPyQ5bQQwEhpl05uJisA/edit>

Need to add label permissions, classes need to be included in require block.

<https://www.postgresql.org/docs/8.3/libpq-pgpass.html>

For Jonathan (with love):

SELECT \* FROM pg\_seclabel;

In controller in container runtime,

Get ip from container

<https://stackoverflow.com/questions/58338051/how-o-get-the-ip-address-of-docker-container-using-docker-sdk-for-python>

Call an external executable to label this IP

Do this before python returns the response to the user

Labeling script:

cops\_platform/container\_runtime/iplabel <ip> <coordinator|instructor|student>

<https://www.postgresql.org/docs/9.1/view-pg-seclabels.html>

IP addresses are labeled by container runtime

Next steps:

send curl commands to get denials

Label appropriate database elements if they are unlabeled, and add rules to the policy