Bonus Activity: Bell-Lapadula Security Model

November 23, 2015

1 Database (and Data) Security: Access Control and the Bell-Lapadula Model

The Bell-Lapadula state machine model is an access control system used in many government and military systems. In this bonus activity, we'll be learning about the principles behind the Bell-Lapadula model. To relate the concept of access control in a manner that should hit close to home, there will also be a quick access control tutorial for use in postgres. Why not sqlite? It turns out that as part of design tradeoffs, sqlite is designed without fine grained access controls.

2 Bell-Lapadula Model

The purpose of the Bell-Lapadula model is to maintain data confidentiality and controlled access to classified information. In the formal model, the entities in a information system are divided into subjects and objects. To relate this to CS145, these subjects could correspond to DB users or different applications that interface with a database, and objects represent tuples/tables in the DB. A data system satsifying the Bell-Lapadula model focuses on the idea of a "secure state", where a certain set of security rules is proved to be true at all times.

A major facet of the Bell-Lapadula model is the presence of "security classifications". These classifications correspond to a heirarchical ranking of how "secure" a particular object is. For example, we will use the labels Top Secret, Secret, Confidential and Unclassified. Each object in our system gets one of these labels. Our goal is to make sure information does not get to those who are not cleared for it. To do so, each user also gets a current and maximal security level that dictates which levels of information they can read/write.

In a Bell-Lapadula information system, only actions that maintain the following three principles can hold. We will use the notation $L_m(s)$ to represent a user s's maximal levels, and $L(O)$ to represent the security level of object O.

1. Simple Security Property: Subject S can read O if $L_m(s) > L(O)$.(No Read Up)
2. Star Property: S can read O iff $L_c(S) \geq L(O)$; S can write O iff $L_c(S) \leq L(O)$ (No Write Down)

3. Discretionary-Security Property: Discretionary Access Control (i.e. the object's owner choosing whether or not specific other users can view their object) is allowed by an access matrix.

The current security level is most importantly used when writing - while a user cannot "write down", this is with respect to their current security level, which is equal to or lower than their maximal security level. A user at a high security level can write down, but they must change their security level first.

Note that a user with a certain security level cannot write data to security levels below them. Why do you think this is?

Explanation Here:

There are a few other details that in the model that you can read more about below.

`https://www.cs.purdue.edu/homes/ninghui/courses/426_Fall10/handouts/426_Fall10_lect21.pdf`
`http://crypto.stanford.edu/~ninghui/courses/Fall03/papers/landwehr_survey.pdf`(section 5.5)

3 Access Control in PostgresSQL

PostgreSQL is a free and open-source DBMS. It also supports features of access control unavailable in sqlite. We'll be going over a few features of the system. If you would like to try out access control in PostGRES, follow one of the following tutorials:

- Mac OSX: `http://www.gotealeaf.com/blog/how-to-install-postgresql-on-a-mac`
- Linux: `https://wiki.postgresql.org/wiki/Detailed_installation_guides`
- Windows: `https://wiki.postgresql.org/wiki/Running_%26_Installing_PostgreSQL_On_Native_Windows`

This tutorial will be based on documentation located at: `http://www.postgresql.org/docs/9.0/static/user-manag.html`

3.1 Basic Roles

```
CREATE ROLE name;
DROP ROLE name;
SELECT rolname FROM pg_roles;
```

The preceding commands are used to create roles and display a list of them.

3.2 Role Attributes

Certain attributes can be appended to the "CREATE ROLE" command to give certain attributes to a user. For example, the command LOGIN can be appended to the command to make the user login when attempting to access the DB.

```
CREATE ROLE name LOGIN;
```

A full list of attributes can be seen here: `http://www.postgresql.org/docs/9.0/static/role-attributes.html`

3.3 Permissions

An owner that owns certain objects in a database can grant certain privileges to users that will allow them to perform certain actions on those objects. For example,

```
GRANT UPDATE ON accounts TO joe;
REVOKE ALL ON accounts FROM PUBLIC;
```

These previous commands (GRANT/REVOKE) are used to give and take away certain privleges. PostgreSQL also allows the user to create user groups.

4 Activity

We will simulate a user set for a simplified commercial application - a grocery store! You will start with two tables that contain important pieces of data about our grocery store. Write more SQL code based on the commands above to generate the users and grant them the appropriate privileges.

- Create Three Users: Boardmen, Cashier and Pricer
- Create a group called "canSeePrices" with the members pricer, cashier and boardmen. Because all of these users should have access to viewing the prices of items in their store, grant the SELECT command on PRICES to the group. View more documentation about groups here: `http://www.postgresql.org/docs/8.1/static/sql-creategroup.html`
- Because neither booardmen or pricers should have control over supermarket orders, grant all privileges on orders to the cashiers.
- The job of a pricer is to measure demand and dictate the prices of the items being sold in the store. Grant pricer update access on PRICES.
- Not only do boardmen add and remove items being sold in the stores, they also have the ability to changes prices. Grant them all privileges on Prices.

```
DROP TABLE IF EXISTS PRICES;
CREATE TABLE PRICES(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
```

```
   PRICE        REAL
);

DROP TABLE IF EXISTS ORDERS;
CREATE TABLE ORDERS(
   NAME           TEXT    NOT NULL,
   ITEMID  INT,
   PRICE REAL
);
#Your Code Here
```