User Authorization for a Hierarchical Account System

Martin Elsman
Zecure

August 11, 2006

Abstract

We present a user authorization technique for a hierarchical account system in terms of a data model for the system and a decision procedure for determining if a given user has a given privilege with given access rights (e.g., read or write access). A user with appropriate privilege rights may create new users, but only with privileges as strong as the user.

1 Introduction

The concept of account naturally emerges when two or more users of a system needs access to the same information. In such systems, it also makes sense to allow a user to have access to more than one account by maintaining a privilege relation between accounts and users.

One problem with such a system is that it is unclear who has the rights to create new accounts and to associate and disassociate users with accounts. A possible extension is thus to enforce all but the top-most account to have a *parent account* and in that sense maintain an account hierarchy. Users with ACCOUNT write access to an account are entitled to create, modify, and delete subaccounts to the account A. The only remaining question is now who should be allowed to modify user privileges and user data such as a user's name. The solution here is to associate each user with exactly one *primary account*. Only users with USER write privileges to a given account should be allowed to modify another user's privileges and data. By enforcing that users may be associated (through privileges) to an account A only if the user's primary account is an ancestor of A, cyclic privilege chains are avoided, which makes authorization straightforward, even in the presence of account privilege inheritance.

1.1 Overview

In Section 2, we present an example account hierarchy, which serves to motivate hierarchical authorization. In Section 3, we describe the data model for a hierarchical authorization system, based on the above ideas. In Section 4, we

```
               c
      b        :
a     :  ----- M1      e
:     : /              :
T --- R1      d ----- M2
        \     :/
         ---- R2
               \
                ----- M3
```

Figure 1: An example account hierarchy.

present a decision procedure for determining if a given user has a given privilege with given access rights (e.g., read or write access). Finally, in Section 6, we conclude.

2 An Example Account Hierarchy

An example account hierarchy may look as shown in Figure 1. In the figure, capitalized names (T, R1, M1, R2, M2, and M3) denote accounts and non-capitalized names (a,b,c,d,e) denote users.

2.1 The Top Administrator (User `a`)

The account `T` is a *top administration account*. The user `a` is *system wide administrator* with full access rights to the account `T` and its subaccounts (recursively). System-wide administration is implemented by giving user `a` the following privilege associations to the account `T`:

```
(T,a) <-> USER(priv=rw,sub=true),
(T,a) <-> ACCOUNT(priv=rw,sub=true)
(T,a) <-> ADMIN(priv=rw,sub=true)
```

These privileges mean that user `a` can access, create, remove, and update account and user information to any depth in the account hierarchy. Moreover, user `a` has access (read and write privileges) to non-account related data such as country and region data.

2.2 The Semi Administrator (User `b`)

Account `R1` is a *semi administrator account* created by user `a`. The user `b` is a *semi administrator* with full access rights to the account `R1` and its subaccounts (recursively). The access privileges are implemented by giving user `b` the following privilege associations to the account `R1`:

```
    (R1,b) <-> USER(priv=rw,sub=true),
    (R1,b) <-> ACCOUNT(priv=rw,sub=true)
```

As a general rule, a user x cannot grant another user y privileges that are stronger than x's privileges. The system we describe here is not capability-based. Thus, revoking a privilege from a user y does not mean that the privilege is revoked from users that have been granted the privilege by the user y.

2.3 The Merchant Account `M1`

The account `M1` is a *merchant account* created by user `b`. User `b` has also created a user `c` with account `M1` as primary account. User `b` has also associated the user `c` with the account `M1`. The access privileges are implemented by user `b` by giving user `c` the following privilege associations to the account `M1`:

```
    (M1,c) <-> USER(priv=rw,sub=false),
    (M1,c) <-> ACCOUNT(priv=rw,sub=false)
```

These privileges do not permit user `c` to create new subaccounts, but they allow user `c` to create new users (for instance with read-only access to account `M1`) and to read and edit account information.

2.4 Comments on User Privileges

Users should always be allowed to change password and perhaps certain kinds of preferences. Other user data, however, such as name, email-address, and so on, for a user x should be editable only by a user y who has `USER(priv=wr)` privileges to x's primary account.

One can ask if users should be allowed to edit their own privileges and account associations? Self-creation, self-activation, and self-strengthening should not be possible. Moreover, to make sure that users can recover most of their user mistakes, self-deletes, self-inactivation, and self-weakening should not be possible either.

3 The Data Model

The data model for the hierarchical authorization system consists of only five entities. An E/R diagram (without attribute annotations) for the data model is given in Figure 3.

In the following subsections, we provide a realization of the data model in terms of SQL `create table` statements for the system.

3.1 Accounts

Here is the data model for accounts:

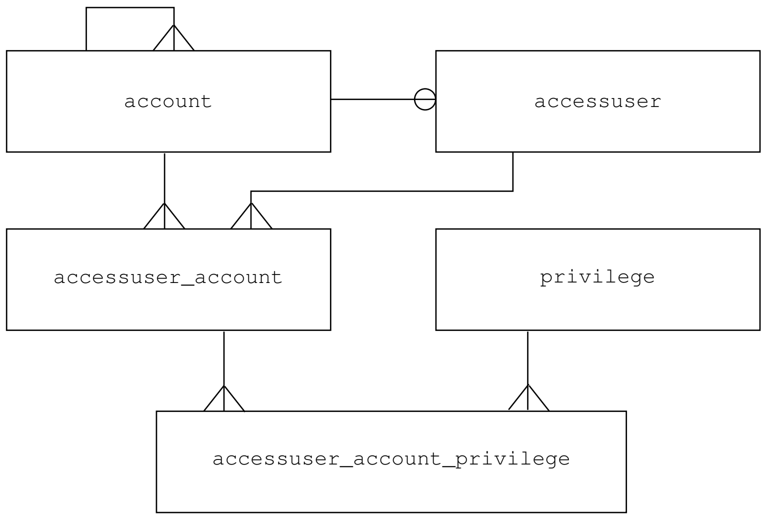

Figure 2: E/R diagram for the hierarchical account system. Boxes in the diagram denotes entities. Forked edges denote one-to-many relationships, where the fork specifies the entity that hosts the foreign key. Similarly, edges annotated with a circle denote one-to-one relationships where the circle specifies the entity that hosts the foreign key.

```
CREATE TABLE account (
    id        SERIAL PRIMARY KEY,
    name      VARCHAR( 80 ) NOT NULL,
    parent_id INTEGER REFERENCES account( id )
);
```

The `parent_id` attribute specifies the parent account of accounts other than the root account (for which the attribute is `null`).

3.2 Users

Here is the data model for users—for historical reasons, the term *access user* is used:

```
CREATE TABLE accessuser (
    id         SERIAL PRIMARY KEY,
    login      VARCHAR( 32 ) UNIQUE NOT NULL,
    passwd     VARCHAR( 64 ), -- null initially; updated with
                              -- password hash digest when
                              -- password is sent to user.
    name       VARCHAR( 100 ) NOT NULL,
    email      VARCHAR( 128 ),
    account_id INTEGER REFERENCES account( id ) -- primary account
);
```

Here the `account_id` attribute denotes the user's primary account. Users may be associated with accounts through the table `accessuser_account`:

```
CREATE TABLE accessuser_account (
    id            SERIAL PRIMARY KEY,
    account_id    INTEGER REFERENCES account( id ) NOT NULL,
    accessuser_id INTEGER REFERENCES accessuser( id ) NOT NULL,
    UNIQUE( accessuser_id,  account_id )
);
```

A property, which is not checked for at the database level, but which is maintained by the application code, is that a user may be associated only with accounts that are descendants of the user's primary account.

3.3 Privileges

```
CREATE TABLE privilege (
    id    SERIAL PRIMARY KEY,
    name  VARCHAR( 64 ) UNIQUE NOT NULL,
    info  TEXT
);
```

We assume built-in privileges USER and ACCOUNT:

```
INSERT INTO privilege (id, name, info)
  VALUES (1, 'USER', 'Administration of users');
INSERT INTO privilege (id, name, info)
  VALUES (2, 'ACCOUNT', 'Administration of subaccounts');
```

Privileges are associated with a user's association with an account through the table `accessuser_account_privilege`:

```
CREATE TABLE accessuser_account_privilege (
    id                    SERIAL PRIMARY KEY,
    privtype              VARCHAR(2) NOT NULL DEFAULT 'r'
                            CHECK (privtype IN ('r','rw')),
    subaccount            BOOLEAN DEFAULT false, -- Allow subaccount access
    accessuser_account_id INTEGER REFERENCES accessuser_account( id )
                            NOT NULL,
    privilege_id          INTEGER REFERENCES privilege( id ) NOT NULL,
    UNIQUE( accessuser_account_id,  privilege_id )
);
```

Here the `privtype` attribute specifies whether the user should have read access or both read and write access to the resource (privilege). The `subaccount` boolean specifies whether the privilege should be inherited by descendant subaccounts.

4 An Authorization Decision Procedure

The following Ruby function checks whether a given user (identified by calling the method `find_accessuser_id`) has certain privileges for a given account:

```
def auth_account(priv,privtype,account,sub)
  accessuser_id = find_accessuser_id
  clauses = ""
  # if the application asks for read privileges (r), it is
  # ok if the user has read-write privileges (rw).
  clauses += " and aap.privtype = 'rw'" if privtype == 'rw'
  clauses += " and aap.subaccount = true" if sub
  joins = ("inner join accessuser_account_privilege as aap" +
           " on privilege.id = aap.privilege_id " +
           "inner join accessuser_account as aa" +
           " on aa.id = aap.accessuser_account_id " +
           "inner join accessuser as a" +
           " on a.id = aa.accessuser_id")
  conditions = [("aa.accessuser_id = ? and aa.account_id = ? " +
                 "and privilege.name = ?" +
                 clauses), accessuser_id, account.id, priv]
  p = Privilege.find( :first, :select => 'privilege.id',
                      :joins => joins, :conditions => conditions )
  return false if p.nil?
  return true
end
```

The following method `auth0` takes three arguments, `priv`, `pt` , and `account`. The method returns `true` if the user has privilege `priv` with privilege type `pt` for account or any ancestor. The method returns `false` otherwise. Examples of `priv` are ACCOUNT and USER. The argument `pt` must be either `'r'` or `'rw'`:

```
def auth0( priv, privtype, account, sub = false )
  return true if auth_account( priv, privtype, account, sub )
  return false if account.parent.nil?
  auth0( priv, privtype, account.parent, true)   # sub = true
end
```

The following method `auth` is the main method to be called by any action that requires authorization. The method takes two arguments `priv` and `pt` and returns `true` if the user has privilege `priv` with privilege type `pt` for the account currently being administered or any ancestor account. The method returns `false` otherwise.

```
def auth( priv, privtype = 'r' )
  auth0( priv, privtype, find_current_account )
end
```

The method `find_current_account` returns the id of the account currently being administered by the user.

5 Related Work

For a discussion about authorization in the context of community sites, see [2]. For a survey of authentication and authorization infrastructures in distributed systems (and in general), see [1].

6 Conclusion

We have presented a data model for a hierarchical authorization system and described an authorization decision procedure for the system. We have not been concerned here about authentication but rather focused on authorization, that is, the question whether a particular user has the necessary privilege rights to perform a particular action.

The hierarchical authorization system allows a user with appropriate privileges to create new users, but only with privileges as strong as the user himself. This property guarantees that a user cannot create users who are more powerful than the user himself.

There are several issues that we have not covered in this note. First, for avoiding excessive database authorization lookup, a simple memorization technique can be deployed and associated with the user's session data. Special care must be taken to insure that this authorization cache is cleared when the user's privileges are altered by an administrator of the user's primary account. Second, we have not given examples of how RubyOnRails filters may be used to run authorization code prior to controller code for specific actions (this is somewhat specific to the RubyOnRails framework). Finally, there are many security issues associated with the notion of authorization that we have not mentioned here. For instance, even if a user is authorized to perform a particular action (e.g., edit account data), it is essential that the application checks that the account to be edited is in the set of accounts that are allowed to be edited by the user. Lack of checking for such properties is a common security breach in many Web based systems.

Acknowledgments

Thanks to Bo Lorentsen at Zecure for many great discussions concerning this work.

References

[1] Jesús Luna García. Authentication and authorization in distributed systems: a survey on aai technologies. Technical report, Polytechnic University of Catalonia, 2004.

[2] Philip Greenspun. *Philip and Alex's guide to Web publishing.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.