Trusted RUBIX™
Version 6

Security Policy Manager Tutorial
Revision 7

RELATIONAL DATABASE MANAGEMENT SYSTEM
Infosystems Technology, Inc.
4 Professional Dr - Suite 118
Gaithersburg, MD 20879
TEL +1-202-412-0152
Infosystems Technology, Inc.
4 Professional Dr - Suite 118
Gaithersburg, MD 20879

THIS DOCUMENTATION CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF INFOSYSTEMS TECHNOLOGY, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF INFOSYSTEMS TECHNOLOGY, INC. FOR FULL DETAILS OF THE TERMS AND CONDITIONS FOR USING THE SOFTWARE, PLEASE REFER TO THE ITI-TRUSTED RUBIX USER LICENSE AGREEMENT.

The information in this document is subject to change without notice and should not be construed as a commitment by ITI.

Infosystems Technology, Inc. assumes no responsibility for any errors that may appear in this document.

*RUBIX*® is a trademark of Infosystems Technology, Inc.

*UNIX*® is a trademark of The Open Group.

*Microsoft*® is a trademark of the Microsoft Corporation.

Printed in U.S.A.
1 OVERVIEW

This tutorial demonstrates the features of the Trusted Rubix Security Policy Manager (SPM) and its basic operations. The TR SPM uses an XML policy language based upon the policy language of the XACML standard. The language is referred to as Trusted RUBIX Security Markup Language (RXSML) and is used to implement an extremely flexible and powerful Attribute Based Access Control (ABAC) security model.

The tutorial uses five examples and the document is intended to be read sequentially. That is, procedures are covered in the earlier sections that are assumed in the latter sections. It is anticipated that the reader will want to refer to the Security Policy Manager Reference Guide for further details. As a minimum, it is recommended that the reader first become familiar with the introduction in the guide for an overview of the RXSML language.

The first example (Section 3) focuses on the procedure to create, apply, update, and delete a simple security policy. Also, it demonstrates global policy features such as MAC override and subtree/node scopes. It also shows the behavior of operating on an SQL view that references a table with an associated SPM policy.

The second example (Section 4) focuses on cross domain policies. The MAC override is used to create policies that allow highly controlled information flows between incomparable MLS domains. The example also demonstrates how to use the set-field obligation to create a “cover story” for a column value.

The third example (Section 5) constructs a white list of IP addresses for table access. It demonstrates how normal row data may be used to control the behavior of a security policy. In this case, a special table is constructed to hold the IP white list which may be updated normally using SQL operations. The SPM is used to tightly control access to the IP white list table.

The forth example (Section 6) uses the SPM to restrict access by RDBMS users to rows they created. It uses the set-field obligation to store the RDBMS user name of the creator of a row into a reserved column as the row is inserted. The reserved column is then used to restrict access to the row. Updating of the reserved column is prohibited.

The last example (Section 7) uses the SPM to restrict access by application users to rows they created. An application user connects to and uses a RDBMS application but may not connect directly to the database. It uses the set-field obligation to store the application user ID of the creator of a row into a reserved column as the row is inserted. The reserved column is then used to restrict access to the row. Updating of the reserved column is prohibited.

Each example gives complete SPM policy code listings and step-by-step instructions for implementing the security behavior. Also, example behavior is demonstrated using the RXISQL database client.
2 HOST AND DATABASE INITIALIZATION

The tutorial uses a common set of users and database objects for many of the examples. This section describes the steps required to initialize the hosts supporting the Trusted Rubix software and the database used for the tutorial. Note that if you do not wish to configure your environment exactly as described in the following steps you will need to use equivalent elements (security labels, user names, and database objects) and make appropriate changes to the RXSML policy configuration files and SQL database scripts presented in this tutorial. Generally, you will need to replace the literal values (such as table names and table columns) in the RXSML and SQL code with values that match those used in your system.

2.1 Host Initialization

A trusted server host must be configured with the TR RDBMS product installed. At least one remote client host should also be configured to execute TR clients and connect to databases on the server host. The client host may be trusted (i.e., run a trusted operating system) or not. If the client is not running a trusted operating system then the network configuration of the trusted server should recognize the client at a fixed sensitivity label of Unclassified. Much of the tutorial may be performed using only the server host.

The OS-MAC policy of the client and server host operating systems used is assumed to have at least the following labels:

- Unclassified (U),
- Confidential Able (C Able), and
- Confidential Baker (C Baker).

Confidential strictly dominates Unclassified. If the client host does not run a trusted operating system it should be recognized as Unclassified by the server host.

Operating system users should be created named spmadmin, spmuser1, and spmuser2. Their group membership is not relevant to the tutorial. Each user should be capable of acquiring a session label of Unclassified (U), Confidential Able (C Able), and Confidential Baker (C Baker). To simplify the tutorial, the spmadmin user should be given all TR authorizations (i.e., rubix.*).

2.2 Database Object Initialization

The database objects should be re-created for each example in the tutorial. The creation of all database objects should be performed by the spmadmin user. Note that the operating system shell prompt of all examples displays the user performing the operation and the current session label (e.g., spmadmin(U)>). The Trusted Rubix RXISQL client will always have a prompt similar to rxsq:1>.
First ensure the *db* database does not exist and then create the *db* database with the following commands:

```
spmadmin(U)> rxd b -df db
spmadmin(U)> rxsql -d master
rxsql:1> create database db;
rxsql:2> q
spmadmin(U)>
```

At this point it is recommended to create all RXSML policy files, add them to the Trusted Policy Repository, and apply them to the database objects. This step is specific to each tutorial example and the exact steps are given during the discussion of each example. In practice, it is advisable to apply policy to objects **before** the object is created to ensure the object is always protected by the policy. The exception to this rule is the database object (i.e., from the create database statement). Because the database object is needed to store policy application mappings, it must be created before any policy may be applied. Note that during this time the database is protected by the DAC and OS-MAC security policies.

Next we create the *cat* catalog, *sch* schema, and *tab* table. To simplify the tutorial, we will give required DAC permissions for all objects to *public*.

```
spmadmin(U)> rxsql -d db
rxsql:1> create catalog cat;
rxsql:2> create schema cat.sch;
rxsql:3> create table cat.sch.tab (name varchar(20), pay_grade integer,
    organization varchar(20));
rxsql:4> grant exec on database to public;
  1 privilege granted.
rxsql:5> grant exec on catalog cat to public;
  1 privilege granted.
rxsql:6> grant exec on schema cat.sch to public;
  1 privilege granted.
rxsql:7> grant all privileges on table cat.sch.tab to public;
  32 privileges granted.
rxsql:8> commit;
rxsql:9> q
spmadmin(U)>
```

The *cat.sch.tab* table represents a simplistic representation of government employees. The table contains three columns *name*, *pay_grade*, and *organization*. Note that the hidden column *rowlabel* also exists.
3 CREATING, EDITING, AND APPLYING A BASIC POLICY

3.1 The user-control Policy

The following is the user-control policy used for this example. To construct the user-control.rxsm file RXSML policy configuration file the following text should be placed into a file named user-control.rxsm. Note that the line numbers are added for convenience and should not be included in the actual policy configuration file. Editing may be performed using any XML editor or an unformatted text editor. A line-by-line explanation is given after the code listing.
Line 1 of the policy contains a standard XML declaration. This declaration should be included at the top of every RXSML policy configuration file.

Lines 2-7 contain the opening <Policy> element for the user-control policy. Because this <Policy> element is the top-level XML element in the file it is special. From an XML perspective it is special because it contains XML namespace and schema declarations.
From an RXSML perspective it is special because its PolicyId attribute (or PolicySetId if this were a PolicySet element) defines the policy name in the Trusted Policy Repository. Also, the top-level <Policy> or <PolicySet> element may contain MacOverride or Scope attributes which have implications for all sub-elements.

Lines 3-5 contain the XML namespace and schema declarations. In general, this should be included in all top-level <Policy> or <PolicySet> elements. In our example, the location of the RXSML schema file (RXSMLSchema.xsd) is assumed to be in the same directory as the RXSML policy configuration file being edited. If it is desired to place the RXSML schema file in a different location then line 5 would need to be changed accordingly.

Line 6 contains the required PolicyId attribute and is set to user-control. This value will define the name of the policy within the Trusted Policy Repository, define the name of the RXSML file that is created when the policy is extracted from the repository (user-control.rxsmf in this case), and define the identifier used to include the policy in policy sets by reference (i.e., using the PolicyIdReference element).

Line 7 contains the required RuleCombiningAlgId attribute and is set to ordered-permit-overrides. This attribute defines how multiple rules are combined to form a single policy decision. For our example, the rules will be evaluated in order and if any rule evaluates to Permit then our policy immediately evaluates to Permit and no other rules are evaluated. The policy will evaluate to Deny only if at least one rule evaluates to Deny and all other rules evaluate to NotApplicable. If all rules evaluate to NotApplicable then our policy will evaluate to NotApplicable.

Lines 9-13 contain an optional, free-form text description of the policy. The description is ignored during evaluation.

Line 15 contains the <Target> element for the policy. The target specifies a set of attributes for which the <Policy> element will apply. If a target does not match a given evaluation environment then the <Policy> element evaluates to NotApplicable. In our case the <Target> element is empty specifying that the <Policy> element is applicable to all evaluation environments.

Lines 17-26 contain the allow-users rule. This rule contains the basic logic for our policy which permits all operations for the spmadmin user and denies all operations for all other users. The Effect attribute of the <Rule> element defines the outcome of the rule if the target is matched by the evaluation environment and the rule’s condition is true. Possible values for the Effect attribute are Permit and Deny. In our case, the rule will evaluate to Permit if the target is matched and the condition is true.

Lines 18-25 contain the target for the allow-users rule. While a target may specify a set of subject, resource, action, and environment attributes our target only specifies a single subject attribute, the spmadmin user. All unspecified attribute categories (resource, action, and environment for the allow-users rule) match all possible attributes. The subject is specified by testing for equality (line 20) between the literal string spmadmin (line 21) and the subject-name subject attribute (line 22).

The allow-users rule contains no Condition attribute; therefore, the condition is implicitly always true. All of the logic for the rule is contained within its target. If the Target element evaluates to Match then the allow-users rule will evaluate to Permit.

Lines 28-31 contain the deny-others rule. This is a simple rule with an empty target and no condition. Therefore it matches all evaluation environments and always has a true condition. If it is evaluated, it will always evaluate to its Effect, Deny in this case. Note that because of the policy’s rule combining algorithm (ordered-permit-overrides) the deny-others rule will only be evaluated if the allows-users rule
does not evaluate to *Permit*. Therefore, this rule will cause the policy to evaluate to *Deny* for all users other than the *spmadmin* user.

The sequence of steps that will occur when evaluating the *user-control* policy follows:

1. Evaluate policy target (line 15): Because it is empty it always evaluates to *Match*.
2. Evaluate *allow-user* rule (lines 17-26):
   
   a. Evaluate rule target (lines 18-25): Evaluates to *Match* if the *subject-name* attribute equals *spmadmin*; otherwise, it evaluates to *NoMatch*. If it evaluates to *NoMatch* the rule evaluation stops immediately and the rule evaluates to *NotApplicable*.
   
   b. Evaluate the rule condition: Because the rule condition is missing, it always evaluates to *true* and the policy evaluates to *Permit*. When the rule evaluates to *Permit* the policy evaluation stops immediately and the policy evaluates to *Permit*.

3. Evaluate *deny-others* rule (lines 28-30):
   
   a. Evaluate the rule target (line 29): Because it is empty it always evaluate to *Match*.
   
   b. Evaluate the rule condition: Because the rule condition is missing, it always evaluates to *true* and the rule will evaluate to *Deny*. Because there are no other rules to evaluate after the *deny-others* rule, the policy will evaluate to *Deny*.

Note that the *user-control* policy is applicable for all evaluation environments. In other words, for any possible evaluation it will reach a policy decision of *Permit* or *Deny* and will never evaluate to *NotApplicable*. It is therefore suitable to control access to an object on its own. If a policy (or policy set) were to have evaluation environments that evaluate to *NotApplicable* then it should be combined with other policies (or policy sets) in a policy set before being used to control access to an object. While a policy evaluation of *NotApplicable* will cause an operation to be denied, it is not good practice to rely on this behavior. Instead, policies should be written that cover all possible evaluation environments.

### 3.2 Database and Policy Initialization

The database should first be initialized as described in Section 2.2 (Database Object Initialization). Note that in practice, the policy should be applied to the RDBMS object(s) before they are created. Next the RXSML policy configuration file *user-control.rsxml* described in Section 3.1 should be created. Note that the policy configuration file may be created and edited on any platform using any XML or plain text editor. Once the RXSML policy configuration file has been created and edited it should be placed on the server host platform with read access given to the *spmadmin* user. In practice, this is the time when the SPM Security Administrator would visually examine the policy configuration file for final verification.

The *user-control.rsxml* policy configuration file will now be added to the Trusted Policy Repository. The Trusted Policy Repository stores verified RXSML policy configuration files in a protected directory so that updating and applying RXSML policies may be tightly controlled. The Trusted Policy Repository stores RXSML policy configuration files in the *RUBIXHOME/etc/security/policies* directory. Individual policy configuration files are named for their top-level *PolicyId* or *PolicySetId* values (in our case *user-control*).
We will first check the syntax of our *user-control* policy by using the `rxpolman -s` administrative command. This command makes syntax checks beyond those made by a schema checking XML editor. It should be noted that even if a policy passes the syntax check it may still produce errors at runtime. The following operations check the syntax of a policy configuration file. Note that if there had been errors in our syntax appropriate error messages with lines numbers would have been displayed. Also note that the syntax is automatically checked when a policy is created or updated.

```bash
spmadmin(U) > rxpolman -s ./user-control.rxsm
RXPOLMAN: Policy in file './user-control.rxsm' has correct syntax

spmadmin(U)
```

Next, we will ensure that there is no current *user-control.rxsm* policy file in the Trusted Policy Repository. Then, we will create the new policy in the repository. This is accomplished using the `rxpolman -c` administrative command. The `rubix.policy.create` authorization is required to add files to the repository. To delete a policy from the repository we use the `rxpolman -d` administrative command. The `rubix.policy.delete` authorization is required to remove files from the repository. Note that if the *user-control* policy already exists in the repository you will receive different output from the `rxpolman -d user-control` operation. Also note that the current working directory is assumed to contain the *user-control.rxsm* file.

```bash
spmadmin(U) > rxpolman -d user-control
RXPOLMAN: Policy 'user-control' does not exist
RXPOLMAN: Delete failed with error 20001 (policy or policy assignment does not exist)

spmadmin(U) > rxpolman -c ./user-control.rxsm
RXPOLMAN: Created policy 'user-control'
RXPOLMAN: Policy created from file './user-control.rxsm'

spmadmin(U)
```

The *user-control* policy may now be applied to RDBMS objects. Once applied to an object, the policy will control access to that object and optionally subordinate objects in the database object tree. Note that the policy may be applied to multiple objects and all applications will reference the same RXSML policy configuration file. This allows the policy behavior of all objects associated with the policy to be updated simply by updating the single RXSML policy configuration file. The policy is applied to an object using the `rxpolman -a` administrative command. The `rubix.policy.apply` authorization is required to apply policies to RDBMS objects. For our example, we are applying the *user-control* policy to the *cat.sch.tab* table object. Note that the user’s session label must equal the sensitivity label of the database.

```bash
spmadmin(U) > rxpolman -a user-control -o db.cat.sch.tab
RXPOLMAN: Operating on DBMS Object: 'db.cat.sch.tab'
RXPOLMAN: Policy 'user-control' assigned to DBMS object

spmadmin(U)
```

We can verify which policies are in the Trusted Policy Repository and which objects they are assigned to using the `rxpolman -l` administrative command. The `rubix.policy.list` authorization is required to list policies in the repository and list policy assignments. This is demonstrated by the following operations.
At this point the `cat.sch.tab` table should be created (please refer back to Section 2.2). Applying the policy before creating the RDBMS object ensures that the object is always under the control of the policy.

3.3 Initial Policy Behavior – Single User Permitted

The `user-control` policy simply permits all operations to the `spmadmin` user and denies all operations for all other users. To demonstrate this we will begin by inserting some records into the `cat.sch.tab` table.

First, the `spmadmin` user:

```plaintext
spmadmin(U)> rxpolman -l
RXPOLMAN: Listing policies:
   'user-control'
RXPOLMAN: Listed 1 policy
RXPOLMAN: Listing was successful
spmadmin(U)> rxpolman -l -o db
RXPOLMAN: Operating on DBMS Object: 'db'
RXPOLMAN: Listing policy assignments for database 'db':
   'user-control' assigned to 'db.cat.sch.tab'
RXPOLMAN: Listed 1 policy assignment
RXPOLMAN: Listing was successful
spmadmin(U>)
```

All operations succeed as expected. Now we will attempt to perform similar operations using the `spmuser1` user:

```plaintext
spmadmin(U)> rxsql -d db
rxsql:1> insert into cat.sch.tab values 'spmadmin', 11, 'NSA';
1 record inserted.
rxsql:2> insert into cat.sch.tab values 'spmuser1', 9, 'CIA';
1 record inserted.
rxsql:3> insert into cat.sch.tab values 'spmuser2', 5, 'DISA';
1 record inserted.
rxsql:4> commit;
rxsql:5> select * from cat.sch.tab;
<table>
<thead>
<tr>
<th>tab.name</th>
<th>tab.pay_grade</th>
<th>tab.organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmadmin</td>
<td>11</td>
<td>NSA</td>
</tr>
<tr>
<td>spmuser1</td>
<td>9</td>
<td>CIA</td>
</tr>
<tr>
<td>spmuser2</td>
<td>5</td>
<td>DISA</td>
</tr>
</tbody>
</table>
3 records displayed.
rxsql:6> q
```
All operations on the `cat.sch.tab` table fail with a “security policy access violation (RA013)” error. Internally to the server, when an SQL operation is executed on the `cat.sch.tab` table, the `cat` catalog and `sch` schema objects are successfully opened. These objects have no SPM policy assigned to them and are therefore only under the DAC and OS-MAC policies. Since these policies are satisfied, the catalog and schema opens are successful. Then an attempt is made to open the `tab` table. This fails because the `spmuser1` user does not have permission according to the `user-access` policy. Similar behavior will occur for any user other than the `spmadmin` user.

### 3.4 Updated Policy Behavior – Adding Permitted Users

We will now update the `user-access` policy to include a new permitted user. While it is possible to edit the `user-access.rxsml` file that exists outside the Trusted Policy Repository, it is good practice to first extract the current policy file from the repository for update. This ensures we are updating the most recent version of the file. We extract a policy file using the `rxpolman -e` administrative command. The `rubix.policy.extract` authorization is required to extract policies from the repository. The policy file will be created with a name equal to the `PolicyId` or `PolicySetId` attribute of the top-level XML element. In our case, it will create a file named `user-control.rxsml` in our current working directory. The file will be read-writable only by the user performing the extraction. Note that there must not be a pre-existing file in the current working directory with the same name. The following operations demonstrate extracting our `user-control` policy.

```
spmadmin(U) > cd /home/spmadmin/policies
spmadmin(U) > rxpolman -e user-control
RXPOLMAN: Policy 'user-control' extracted to
'/home/spmadmin/policies/user-control.rxsml'
RXPOLMAN: Policy 'user-control' extracted to current working directory
```

We will now edit the `user-control.rxsml` file to add a new permitted user. A code listing of the new policy configuration file follows.
<?xml version="1.0" encoding="UTF-8"?>
<Policy
xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicyId="user-control"
RuleCombiningAlgId="ordered-permit-overrides">

<Description>
  For all operations:
  Allow users included in the allow-users rule target (only spmadmin).
  Deny all other users.
</Description>

<Target/>

<Rule RuleId="allow-users" Effect="Permit">
  <Target> <Subjects>
    <Subject>
      <SubjectMatch MatchId="equal">
        <AttributeValue DataType="string">spmadmin</AttributeValue>
        <SubjectAttributeDesignator AttributeId="subject-name"/>
      </SubjectMatch>
    </Subject>
    <Subject>
      <SubjectMatch MatchId="equal">
        <AttributeValue DataType="string">spmuser1</AttributeValue>
        <SubjectAttributeDesignator AttributeId="subject-name"/>
      </SubjectMatch>
    </Subject>
  </Subjects>
</Rule>

<Rule RuleId="deny-others" Effect="Deny">
  <Target/>
</Rule>

</Policy>
The new policy file is the same as the original with lines 25-30 added. This adds a new `<Subject>` element to our `<Subjects>` element in the target. In order for a `<Subjects>` element to evaluate to `Match` at least one of its `<Subject>` elements must evaluate to `Match`. So the new target behavior will evaluate to `Match` if the `subject-name` is `spmadmin` or `spmuser1`.

We update the policy in the repository using the `rxpolman -u` administrative command. The `rubix.policy.update` authorization is required to update policies in the repository.

```
spmadmin(U)> cd /home/spmadmin/policies
spmadmin(U)> rxpolman -u ./user-control.rxsml
RXPOLMAN: Updated policy 'user-control'
RXPOLMAN: Policy updated from file './user-control.rxsml'
```

The policy behavior has now been updated and the behavior will become effective with the next database transaction that starts after the policy update was performed. Note that if other policy sets were referencing the `user-control` policy their behavior would also be updated.

We can verify the new policy behavior by performing a select on the `cat.sch.tab` table using the `spmuser1` user as follows.

```
spmuser1(U)> rxisql -d db
rxsql:1> select * from cat.sch.tab;
tab.name | tab.pay_grade | tab.organization
---------- | -------------- |---------------------
spmadmin  |            11  | NSA
spmuser1  |             9  | CIA
spmuser2  |             5  | DISA
3 records displayed.
rxsql:2> q
```

As expected the `spmuser1` has been permitted access to the `tab` table.

### 3.5 Updated Policy Behavior – Hiding the Existence of Objects

We will now update the `user-control` policy so that it will hide the existence of objects for which a user is not permitted to access. This is accomplished using the `set-error-code` obligation. An obligation is an action that is taken in concert with a policy decision. The execution of an obligation may be predicated upon a `Permit` or `Deny` outcome of the policy containing the obligation. In our case, we wish to execute the obligation if the policy has a `Deny` outcome. A code listing of the new policy configuration file follows.
<?xml version="1.0" encoding="UTF-8"?>
<Policy
    xmlns="http://www.rubix.com/2008/RXSMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    PolicyId="user-control"
    RuleCombiningAlgId="ordered-permit-overrides">
  <Description>
    For all operations:
    Allow users included in the allow-users rule target (only spmadmin).
    Deny all other users.
  </Description>

  <Target/>

  <Rule RuleId="allow-users" Effect="Permit">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="equal">
            <AttributeValue DataType="string">spmadmin</AttributeValue>
            <SubjectAttributeDesignator AttributeId="subject-name"/>
          </SubjectMatch>
        </Subject>
        <Subject>
          <SubjectMatch MatchId="equal">
            <AttributeValue DataType="string">spmuser1</AttributeValue>
            <SubjectAttributeDesignator AttributeId="subject-name"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
    </Target>
  </Rule>

  <Rule RuleId="deny-others" Effect="Deny">
    <Target/>
  </Rule>

  <Obligations>
    <Obligation FulfillOn="Deny" ObligationId="set-error-code">
      <ErrorCode ErrorCodeId="object-not-found"/>
    </Obligation>
  </Obligations>
</Policy>
The new `user-control.rxsm` policy file differs from the previous version only in the addition of the `<Obligations>` element of lines 38-40. Within the `<Obligations>` element we have a single `<Obligation>` that will be executed when the enclosing policy evaluates to `Deny`. The particular action taken by the obligation is specified by the `ObligationId` attribute, `set-error-code` in our case. Obligations never change the `Permit/Deny` behavior of a policy, so our policy will still only allow `spadmin` and `spmuser1` access to the `cat.sch.tab` table. The change in behavior we will see is the SQL error code returned when access is denied to the table. The new policy may be added using the `rxpolman –u` command, just as in the previous section. The new policy behavior is shown with the following commands. Note that they are executed by the `spmuser2` user.

```
spmuser2(U)> rxisql -d db
rxjsql:1> select * from cat.sch.tab;
select * from cat.sch.tab
   ^
named relation is inaccessible or non-existent (RR005)
rxjsql:2> q
spmuser2(U>)
```

Note that the error message returned is now “named relation is inaccessible or non-existent (RR005)” where before we added the obligation a denied user would receive an error message of “security policy access violation (RA013).” This is useful when security requirements dictate that the existence of an object be hidden from users.

### 3.6 Updated Policy Behavior – Controlling a Subtree

We will now change the behavior of the policy so that it controls access to an entire subtree of RDBMS objects. **Trusted Rubix** database objects are organized as an inverted tree. At the root of the tree is the database, the database contains catalogs, catalogs contain schemata, and schemata contains views, tables, and indexes. Currently our `user-control` policy is assigned to and controlling only the `cat.sch.tab` table. The scope of a policy is controlled through the `PolicyScope` attribute of the top-level `<Policy>` or `<PolicySet>` element. The `PolicyScope` attribute may be `Node`, which is the default, or `Subtree`. When the `PolicyScope` is set to `Node` then the policy only controls behavior of the object to which it is assigned. If the `PolicyScope` is set to `Subtree` then it controls behavior for all subordinate objects that have no associated policy. When an object has multiple ancestor objects with `Subtree` scoped policies, the closest ancestor is chosen.

To set our `user-control` policy to have a `PolicyScope` of `Subtree` we change the `<Policy>` element as follows. The remaining code in the policy remain the same.
The new PolicyScope attribute value is specified on line 7. The new policy may be updated using the rxpolman –u administrative command. While the user-control policy is assigned to the cat.sch.tab table no difference in behavior will be noticed. This is because the tab table cannot have subordinate objects. However, if we assign the new user-control policy to the cat.sch schema then the sch schema and all objects inside the schema (including the tab table) will be protected by the policy. We can accomplish this as follows.

```
spmadmin(U)> rxpolman -a user-control -o db.cat.sch
RXPOLMAN: Operating on DBMS Object: 'db.cat.sch'
RXPOLMAN: Policy 'user-control' assigned to DBMS object
spmadmin(U)> rxpolman -a NULL -o db.cat.sch.tab
RXPOLMAN: Operating on DBMS Object: 'db.cat.sch.tab'
RXPOLMAN: Assignment removed from DBMS Object
RXPOLMAN: Policy 'NULL' assigned to DBMS object
spmadmin(U)>
```

We first assign the new policy to the cat.sch schema object. Next, we remove the assignment from the cat.sch.tab table by assigning the special “NULL” policy. Note that since we first apply the policy to the schema and then remove it from the table the table is always protected by the policy. Had we first removed it from the table then there would be a period of time when the table was not under the policy.

When we now attempt to select from the table as spmuser2 and we receive the following results,

```
spmuser2(U)> rxisql -d db
rxisql:1> select * from cat.sch.tab;
select * from cat.sch.tab
^  
named schema is inaccessible or non-existent (3F001)
rxisql:2> q
spmuser2(U)>
```

note that the error message now indicates the schema does not exist whereas in the previous example of the policy behavior on the previous page, the error message indicated the table did not exist. This is because we moved the policy to the schema and it now covers the cat.sch schema object. Internally the catalog is successfully opened but the open on the schema fails. Because the schema is not opened, no operations may be executed on any object inside the schema. This includes our tab table and any new objects that are created inside the schema. This provides a powerful mechanism to create default policy
behavior for new objects. This behavior is demonstrated with the following operations performed by the spmuser1 user.

```
spmuser1(U)> rxsql -d db
rxsql:1> create table cat.sch.MyTable (MyData varchar(50));
rxsql:2> insert into cat.sch.MyTable values 'Only spmadmin and spmuser1 can see this!';
1 record inserted.
rxsql:3> commit;
rxsql:4> q
spmuser1(U>)
```

The spmuser1 is permitted to create the table in the cat.sch schema, as expected. The user-control policy is automatically associated with the new MyTable table because the schema’s policy has a PolicyScope of Subtree, as noted in the example at the top of the previous page. The following operations demonstrate that the spmuser2 is denied access to the new table. Note that we did not make any new policy assignments after we created the MyTable table.

```
spmuser2(U)> rxsql -d db
rxsql:1> select * from cat.sch.MyTab;
select * from cat.sch.MyTab
 name schema is inaccessible or non-existent (3F001)
rxsql:2> q
spmuser2(U>)
```

As expected the spmuser2 user is completely denied access to the cat.sch schema and any object it contains. But the spmadmin user is permitted to select from the table, as follows.

```
spmadmin(U)> rxsql -d db
rxsql:1> select * from cat.sch.MyTable;
MyTable.MyData
--------------------------------------------------
Only spmadmin and spmuser1 can see this!
1 record displayed.
rxsql:2> q
spmadmin(U>)
```

The user-control policy could be assigned to the catalog or database to increase its scope of control. Default policy for an entire database may be enabled simply by assigning subtree scoped policy to the database object. It should be noted that while it is possible to put all policy for all database objects into a single subtree scoped policy configuration file and then assign it to the database, this is discouraged for all but very simple policies. This is because there is a performance benefit from partitioning policy logic into separate configuration files and assigning them to more discrete objects. Also, policy logic that is partitioned into separate configuration files eases modular policy construction and administration.
3.7 Updated Policy Behavior – MAC Override

Our last modification to the user-control policy will be to give it MAC override permission. When a policy has MAC override permission the underlying MAC policy will effectively be “turned off” and the SPM policy will be relied upon for security behavior. It should be noted that new objects (i.e., through a create or insert operation) will always be labeled with the session label irrespective of whether the policy has MAC override permission. Before we add MAC override permission we will first demonstrate some MAC behavior without override enabled. Note that the following operations are performed with a session label of Confidential Able.

```
spadmin(C Able)> rxisql -d db
rxsql:1> insert into cat.sch.tab values ‘ConUser’, 12, ‘CIA’;
1 record inserted.
rxsql:2> commit;
rxsql:3> select tab.rowlabel, tab.* from cat.sch.tab;
tab.rowlabel | tab.name | tab.pay_grade | tab.organization
--------------|----------|---------------|------------------
U | spmadmin | 11 | NSA
U | spmuser1 | 9 | CIA
U | spmuser2 | 5 | DISA
C ABLE | ConUser | 12 | CIA
4 records displayed.
rxsql:4> q
spadmin(C Able)>
```

Note that we now have a row with a C Able row label. While this is visible from a session label of C Able, it is filtered out by the MAC policy when a user selects from a session label of U, as demonstrated below.

```
spadmin(U)> rxisql -d db
rxsql:1> select tab.rowlabel, tab.* from cat.sch.tab;
tab.rowlabel | tab.name | tab.pay_grade | tab.organization
--------------|----------|---------------|------------------
U | spmadmin | 11 | NSA
U | spmuser1 | 9 | CIA
U | spmuser2 | 5 | DISA
3 records displayed.
rxsql:2> q
spadmin(U)>
```

We will now add the MAC override permission for the user-control policy. This is accomplished by modifying the MacOverride attribute of the top-level <Policy> or <PolicySet> element. The MacOverride attribute may take a value of false, the default, or true. We will set it to true by modifying the <Policy> element as follows. Note that the remaining policy code does not change.
The policy should be edited and updated in the repository. The MacOverride attribute is on line 8. The user-control policy will now cause operations to execute with MAC override permission as demonstrated by the following operations.

```
spmadmin(U)> rxisql -d db
rxsql:1> select tab.rowlabel, tab.* from cat.sch.tab;

<table>
<thead>
<tr>
<th>tab.rowlabel</th>
<th>tab.name</th>
<th>tab.pay_grade</th>
<th>tab.organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>spmadmin</td>
<td>11</td>
<td>NSA</td>
</tr>
<tr>
<td>U</td>
<td>spmuser1</td>
<td>9</td>
<td>CIA</td>
</tr>
<tr>
<td>U</td>
<td>spmuser2</td>
<td>5</td>
<td>DISA</td>
</tr>
<tr>
<td>CABLE</td>
<td>ConUser</td>
<td>12</td>
<td>CIA</td>
</tr>
</tbody>
</table>
```

4 records displayed.

rxsql:2> q

Note that the spmadmin user is operating with a session label of U and the select returned the CABLE row. Care should be taken when constructing policies with MAC Override permissions. The MAC access control policy is turned off for all operations on objects covered by the policy; therefore, thought must be given to the desired behavior of all operations on these objects and the SPM policy must constructed accordingly.

### 3.8 Relationship Between Policy and Views

Views are essentially named, stored select statements on underlying tables. Views do not contain rows as tables do but merely provide a pre-configured way of looking at the rows of tables they reference. The set of rows that a view may operate on is completely dependent upon the policies of the referenced tables. SPM policy may only directly control the creating, opening, and dropping of a view object. The SPM policies on the underlying tables control which rows a view may access and whether the underlying table may be opened. This means that once a policy is established on a table, there is no danger that a view may circumvent that policy.

To demonstrate this we will create a view over the `cat.sch.tab` table. Note that the view object does not reside in the `sch` schema and has no SPM policy associated with it.
Note that the MAC override behavior of name_view’s underlying cat.sch.tab table applies when we select from the view. Now we attempt to select using spmuser2.

Note that the spmuser2 is denied access just as they would when selecting from the cat.sch.tab table. Internally, the select of the view causes a select to be initiated on the underlying table. When the underlying table is opened the operation fails as access to the sch schema is denied.

### 3.9 Deleting the user-control Policy

We will now demonstrate how to delete the user-control policy from the Trusted Policy Repository. This is accomplished using the rxpolman –d administrative command. The rubix.policy.delete authorization is required to delete policies from the repository. We will attempt to delete the policy as shown below.

The delete has failed because it is still assigned to the db.cat.sch object (see top example on page 14). We must first remove the policy assignment and then we may delete the policy.
The user-control policy is no longer in the repository and the database objects are now not under an SPM policy but are still protected by the DAC and MAC security policies.
4 CROSS DOMAIN RELEASABILITY POLICY

This example will demonstrate a simple cross domain releasability policy. In this example we will have two distinct security domains, domain1 and domain2. We will use the OS-MAC MLS policy to provide the basic separation between domain1 and domain2 and use the SPM to write highly specific rules to allow controlled information flows between the two domains. Note that we alternatively could have used the Type Enforcement (OS-MAC TE) policy of an SELinux enabled operating system to provide the basic domain separation.

The requirements of the security behavior are:

- Two domains: domain1 and domain2
- domain1 statically connected from the Confidential Able label
- domain2 statically connected from the Confidential Baker label
- Assume that spadmin and spmuser1 are from domain1 and spmuser2 is from domain2
- Single table in domain1 containing US government employees with three columns: name, pay_grade, organization
- Read-write access by domain1 to all rows of table
- Read-only access by domain2 to a subset of rows in the table where pay_grade is less than ten
- Users from domain2 must not see actual values of organization (e.g., NSA, CIA, etc) but shall see a “cover” of US Government
- The rows domain2 sees must be audited
- No access by domain2 to remainder of rows in the table where pay_grade is greater than or equal to 10

4.1 Database Object Initialization

The database objects will be initialized from domain1. The creation of the db database and the application of the policies should be performed by the spadmin user operating at a session label of Unclassified. The creation of the cat catalog, sch schema, tab table, and the rows in the table should be performed by the spadmin user operating at a session label of Confidential Able. Note that this database object initialization is the same as the previous example in Section 3.2 except that some objects are created at Confidential Able. First ensure the db database does not exist, create the db database, and grant database privileges with the following commands:

```
spmadmin(U)> rxdb -df db
spmadmin(U)> rxisql -d master
rxisql:1> create database db;
rxisql:2> q
spmadmin(U)> rxisql -d db
rxisql:1> grant all privileges on database to public;
5 privileges granted.
spmadmin(U>)
```
Next we create the *cat* catalog, *sch* schema, and *tab* table. Note that we use a session label of *Confidential Able*. To simplify the tutorial, we will give required DAC permissions for all objects to *public*. In general it is recommended to create and add policy before the objects are created but to simplify the presentation we will forgo creating and adding the policy until the next step.

```
spmadmin(C Able)> rxisql -d db
rxsql:1> create catalog cat;
rxsql:2> create schema cat.sch;
rxsql:3> create table cat.sch.tab (name varchar(20), pay_grade integer, organization varchar(20));
rxsql:4> insert into cat.sch.tab values 'spmadmin', 11, 'NSA';
1 record inserted.
rxsql:5> insert into cat.sch.tab values 'spmuser1', 9, 'CIA';
1 record inserted.
rxsql:6> insert into cat.sch.tab values 'spmuser2', 5, 'DISA';
1 record inserted.
rxsql:7> grant exec on catalog cat to public;
1 privilege granted.
rxsql:8> grant exec on schema cat.sch to public;
1 privilege granted.
rxsql:9> grant all privileges on table cat.sch.tab to public;
32 privileges granted.
rxsql:10> commit;
rxsql:11> q
spmadmin(C Able)>
```

The database objects are now initialized and we may proceed to creating and adding policy.
4.2 Policy Initialization

This example will introduce modular policy construction. Our global policy sets will consist of smaller policies, each in their own policy configuration file.

4.2.1 The deny and mac-check Policies

To start, we will create two very simple policies, the deny policy and the mac-check policy. Note that these “building block” policies may be used in any number of policy sets.

The deny policy simply always evaluates to Deny. It is used as a “catch-all” to deny access to an object when other policies in the policy set did not explicitly permit access. Its use will become clear as the tutorial progresses. The listing of the deny policy follows.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Policy
3    xmlns="http://www.rubix.com/2008/RXSMLSchema"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://www.rubix.com/2008/RXSMLSchema RXSMLSchema.xsd"
6    PolicyId="deny"
7    RuleCombiningAlgId="ordered-deny-overrides">
8
9    <Description>
10       Deny all operations
11    </Description>
12
13    <Target/>
14
15    <Rule RuleId="deny-all" Effect="Deny">
16       <Target/>
17    </Rule>
18  </Policy>
```
The *deny* rule matches all evaluation environments and has a single rule that always evaluates to *Deny*. Note that a rule with no condition always evaluates to its *Effect* and that an empty target always evaluates to *Match*.

The *mac-check* rule performs an access check for the current operation and object against the operating system’s OS-MAC security policies (MLS and TE). The policy will evaluate to *Permit* if the current operation is permitted according to the OS-MAC policy. If the OS-MAC policy would deny the operation then the *mac-check* policy evaluates to *NotApplicable*.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicyId="mac-check"
RuleCombiningAlgId="ordered-permit-overrides">
  <Description>
    Perform an OS-MAC check
    Permit if MAC check passes; Not Applicable otherwise
  </Description>

  <Target/>

  <Rule RuleId="mac-check-pass" Effect="Permit">
    <Target/>
    <Condition>
      <Apply FunctionId="MAC-check"/>
    </Condition>
  </Rule>
</Policy>
```
The targets of both the policy (line 14) and the rule (line 17) are empty, specifying that they both match all evaluation environments. The *mac-check-pass* rule (lines 16-21) has a single function in its condition, the *MAC-check* function (line 19). Note that if the condition evaluates to *false* the rule will evaluate to *NotApplicable*.

The *MAC-check* function automatically performs an access check against the operating system’s OS-MAC security policy for the current operation and current object. The function returns *true* if the operation would be permitted by the OS-MAC policy and *false* otherwise. The *MAC-check* function also sets an appropriate error code for the operation when the OS-MAC policy would deny the operation. The error code is consistent with the OS-MAC information flow requirements in that it will hide the existence of objects that are not dominated by the session label.

The *deny* and *MAC-check* policies as shown in the listings should be placed into policy configuration files and added to the repository with the *rxpolman –c* administrative command.

### 4.2.2 The *xdomain-open* and *xdomain-select* Policies

The next two policies control the behavior for the two operations that *domain2* may perform in *domain1*, open and select. Our security behavior requirements specify that *domain2* be able to select certain rows from the *cat.sch.tab* table in *domain1*. To accomplish this the *domain2* users will need to open all database objects in the path to the *tab* table and to perform the select. Because the sensitivity label of the *domain1* objects (*C Able*) is incomparable with the session label of *domain2* (*C Baker*) the MAC override permission must be given to *domain2*.

The *xdomain-open* policy will permit *domain2* to open the required objects (*db database, cat schema, sch schema, and tab table*).

The *xdomain-select* policy will permit *domain2* to select the allowable rows.

Both policies will evaluate to *NotApplicable* if the current operation is not *domain2* performing an open (*xdomain-open*) or select (*xdomain-select*).

A listing of the *xdomain-open* policy follows.
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicyId="xdomain-open"
RuleCombiningAlgId="ordered-permit-overrides">

<Description>
Allow domain2 user to open domain1 object
Not Applicable for all others
</Description>

<Target>
<Subjects>
<Subject>
<SubjectMatch MatchId="equal">
<AttributeValue DataType="label">C Baker</AttributeValue>
<SubjectAttributeDesignator AttributeId="session-label"/>
</SubjectMatch>
</Subject>
</Subjects>

<Actions>
<Action>
<ActionMatch MatchId="equal">
<AttributeValue DataType="string">open</AttributeValue>
<ActionAttributeDesignator AttributeId="action-type"/>
</ActionMatch>
</Action>
</Actions>
</Target>

<Rule RuleId="allow-open" Effect="Permit">
<Target/>
</Rule>
</Policy>

All of the policy logic for the xdomain-open policy is in its target (lines 14-28). The target matches an evaluation environment with a session label of C Baker (lines 16-19) and an operation type of open (lines 23-26). If the target matches the evaluation environment then the allow-open rule of lines 30-32 is evaluated which always evaluates to Permit. Note that the allow-open rule’s target is empty, which matches all evaluation environments, and the condition is missing, which implicitly evaluates to true.

A listing of the xdomain-select policy follows.
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicyId="xdomain-select"
RuleCombiningAlgId="ordered-permit-overrides">
  <Description>
    Allow domain2 user restricted select access to domain2 table.  
    Only see rows with pay_grade less than 10  
    The organization field will be set to "US Government"  
    Rows selected will be audited including the name column
  </Description>
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="equal">
          <AttributeValue DataType="label">C Baker</AttributeValue>
          <SubjectAttributeDesignator AttributeId="session-label"/>
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Actions>
      <Action>
        <ActionMatch MatchId="equal">
          <AttributeValue DataType="string">row-select</AttributeValue>
          <ActionAttributeDesignator AttributeId="action-id"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
  <Rule RuleId="allow-select" Effect="Permit">
    <Target/>
    <Condition>
      <Apply FunctionId="less-than">
        <AttributeSelector RequestContextPath="db.cat.sch.tab.pay_grade" DataType="integer"/>
        <AttributeValue DataType="integer">10</AttributeValue>
      </Apply>
    </Condition>
  </Rule>
  <Obligations>
    <Obligation FulfillOn="Permit" ObligationId="set-field">
      <AttributeValue DataType="string">db.cat.sch.tab.organization</AttributeValue>
    </Obligation>
    <Obligation FulfillOn="Permit" ObligationId="audit">
      <AttributeSelector RequestContextPath="db.cat.sch.tab.name" DataType="string"/>
    </Obligation>
  </Obligations>
</Policy>
The xdomain-select policy contains the bulk of the logic for our security behavior. The policy target matches an evaluation environment with a session label of C Baker (lines 18-21) and an operation of row-select (lines 25-28). If the policy target is matched then the allow-select rule (lines 32-40) is evaluated; otherwise, the policy evaluates to NotApplicable. The allow-select rule evaluates to Permit if the db.cat.sch.tab.pay_grade column value (as specified by the <AttributeSelector> of line 36) is less than a value of 10. Internally, this policy will be evaluated against every possible row in a result set and rows that do not result in a policy evaluation of Permit will be filtered out. Thus, a select from the C Baker session label will result only in rows whose pay_grade is less than 10.

The xdomain-select policy also has two obligations (lines 42-51). Both obligations are executed only if the xdomain-select policy evaluates to Permit and the top-level policy set evaluates to Permit. The is specified by the FulfillOn attribute of the <Obligation> elements. Note that if a select operation occurs where our xdomain-select policy is not applicable (e.g., from the C Able session label) then the obligations will not be executed. Thus, an obligation’s execution is intimately tied to the logic of the containing policy.

The first obligation, a set-field obligation (lines 43-46), will set the value of the organization column of each row that is in the permitted result set. Conceptually, a set-field obligation will set the column value after the row has been read (for select operations) and before the row has been written (for insert and update operations). Note that if a select on a table is performed as part of an SQL insert or update statement, our set-field obligation will still be executed. The set-field obligation takes two arguments. The first specifies the column whose value is to be set. In our case the <AttributeValue> element of line 44 specifies the db.cat.sch.tab.organization column value. The second argument is the actual value, the literal value “US Government.” Note that the second argument need not be a simple literal value. For example, it may be calculated from RXSML functions or be one of the attribute selector elements.

The second obligation, an audit obligation (lines 48-50), will create an audit record. The audit record will always contain a default set of attributes, such as the subject name, operation type, and session label. Optionally, a customized string value may be included in the audit record. This is accomplished by adding a single, optional argument to the audit obligation, in our case the value of the db.cat.sch.tab.name column (line 49). Note that multiple values may be easily combined into a single value using the concatenate function.

The xdomain-open and xdomain-select policies as shown in the listings should be placed into policy configuration files and added to the repository with the rxpolman –c administrative command.

4.2.3 The open-obj and table-obj Policy Sets

We now introduce the policy set construct to our tutorial. Policy sets combine the outcome of multiple policies or policy sets into a single outcome. Policies and policy sets may be explicitly included or referenced by identifier. In our case, we will be referencing our previously defined policies by their identifier. Policy sets may also contain obligations, but may not contain any rules. The policies and policy sets are combined according to a policy combining algorithm, which is very similar to a rule combining algorithm.

Our example includes two policy sets, the open-obj and table-obj policy sets. Both policy sets are top-level and will be assigned to objects. Both contain the MAC override permissions needed for cross level operations, as seen by the MacOverride attributes of the <PolicySet> elements. Note that none of the previously defined policies have the MAC override permission. The MAC override permission is only meaningful for a top-level policy or policy set and applies to all policies under it. The open-obj policy set contains all of the logic for opening the database, catalog, and schema objects. The table-obj policy set
contains all of the logic to open and select from the table. Note that the logic in both policy sets covers both the inter-domain case and the intra-domain case.

A listing of the open-obj policy set follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet xmlns="http://www.rubix.com/2008/RXSMLSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicySetId="open-obj"
MacOverride="true"
PolicyCombiningAlgId="ordered-permitoverrides">
<Description>
Logic for opening an object
First, permit if passes MAC check
Second, permit if allowable cross domain context
Third, deny all others
</Description>
<Target/>
<PolicyIdReference>mac-check</PolicyIdReference>
<PolicyIdReference>xdomain-open</PolicyIdReference>
<PolicyIdReference>deny</PolicyIdReference>
</PolicySet>
```

The MAC override permission is specified on line 7. The policy set uses an ordered-permit-overrides policy combining algorithm. This causes the policies to be evaluated in order and the policy set immediately evaluates to Permit if any included policy evaluates to permit. Note that each included policy, other than the deny policy, either evaluates to Permit or NotApplicable. Therefore, the deny policy will only be evaluated if all other policies evaluate to NotApplicable. Because the deny policy always evaluates to Deny, it causes our policy set to evaluate to Deny if no other policy is applicable. The open-obj policy set has an empty target (line 15) so it matches all evaluation environments. It will first evaluate the mac-check policy (line 16). If the operation passes the MAC check (i.e., it is an intra-domain operation) then our policy set evaluates to Permit; otherwise, the xdomain-open policy (line 17) is evaluated. If the xdomain-open policy evaluates to Permit (i.e., it is an inter-domain open operation) then our policy set evaluates to Permit; otherwise, the deny policy (line 18) will be evaluated and our policy set will evaluate to Deny.

The open-obj policy set as shown in the listing should be placed into a policy configuration file and added to the repository with the rxpolman –c administrative command. It should then be assigned to the db database, db.cat catalog, and db.cat.sch schema objects using the rxpolman –a administrative command.
Note that we could have used a scope of *Subtree* for this policy set and then simply assigned it to the *db* database object. However, this would give *domain2* open operation access to all of *domain1*’s object.

A listing of the *table-obj* policy set follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicySetId="table-obj"
MacOverride="true"
PolicyCombiningAlgId="ordered-permit-overrides">
<Description>
Logic for table object
First, permit if passes MAC check
Second, permit if allowable cross domain select
Third, permit if allowable cross domain open
Fourth, deny all others
</Description>
<Target/>
(PolicyIdReference>mac-check</PolicyIdReference>
(PolicyIdReference>xdomain-select</PolicyIdReference>
(PolicyIdReference>xdomain-open</PolicyIdReference>
(PolicyIdReference>deny</PolicyIdReference>
</PolicySet>
```

The *table-obj* policy set is very similar to the previously discussed *open-obj* policy set. The only difference is the addition of the *xdomain-select* policy. Note that including policies by reference allows for efficient code reuse as our *mac-check, xdomain-open, and deny* policies are used in both policy sets. It may also be noted that the *table-obj* policy set will work properly as a substitute for the *open-obj* policy set. This is because the *xdomain-select* policy will always evaluate to *NotApplicable* for operations on databases, catalogs, and schemata. Thus, we could only have a single policy set for our security behavior. For our simple example this is attractive. In practice care must be taken to avoid excessive complexity in the policy logic especially when additional logic may be added in the future.

The *table-obj* policy set as shown in the listing should be placed into a policy configuration file and added to the repository with the *rxpolman –c* administrative command. It should then be assigned to the *db.cat.sch.tab* table object using the *rxpolman –a* administrative command. Note that applying policy to a database must be performed with a session label equal to the database’s object label, *Unclassified* in this case.
4.3 Policy Behavior

The expected behavior of our example will be for users from domain1 (spmadmin and spmuser1 operating at C Able) will be able to read/write all rows in the cat.sch.tab table while users from domain2 (spmuser2 operating at C Baker) will be able to read a subset of the rows (where pay_grade < 10). Furthermore, for domain2 users the organization column will be statically set to “US Government” and the rows accessed will be audited.

First, we shall demonstrate the domain1 behavior by selecting from the table using spmuser1 at a session label of C Able.

```
spmuser1(C Able)> rxisql -d db
rxsql:1> select rowlabel, tab.* from cat.sch.tab;
```

```
tab.rowlabel | tab.name | tab.pay_grade | tab.organization
-------------|----------|---------------|------------------
C ABLE       | spmadmin | 11            | NSA              
C ABLE       | spmuser1 | 9             | CIA              
C ABLE       | spmuser2 | 5             | DISA             
3 records displayed.
```

As expected all rows are accessible to the domain1 user. We now demonstrate the cross domain behavior by selecting from the table using a domain2 user, spmuser2 at C Baker.

```
spmuser2(C Baker)> rxisql -d db
rxsql:1> select rowlabel, tab.* from cat.sch.tab;
```

```
tab.rowlabel | tab.name | tab.pay_grade | tab.organization
-------------|----------|---------------|------------------
C ABLE       | spmuser1 | 9             | US Government    
C ABLE       | spmuser2 | 5             | US Government    
2 records displayed.
```

When the spmuser2 user selects from the table they only receive rows where pay_grade is less than 10 and the organization column has been set to “US Government” (above). This matches our desired security behavior. The rows selected were also audited as shown below. The audit records contain the standard audit data and the “Optional Data” field contains the value of the name column as expected. Note that the spmadmin user must perform the rxauditrpt operation at a label that dominates the audit records, preferably system high.
When a policy is created for a select operation it controls the reading of records from any SQL operation, including the insert operation as demonstrated below.
Here we have created a duplicate catalog, schema, and table, each with a `d2_` prefix. We then inserted all available rows from `cat.sch.tab` into `d2_cat.d2_sch.d2_tab`. Note that only rows permitted for `domain2` to read from `cat.sch.tab` were inserted into `d2_cat.d2_sch.d2_tab`. Also note that the `organization` column has been set to “US Government.” While not shown, the SQL insert operation also produced audit records for the select from `cat.sch.tab`. Similar behavior would occur if the `cat.sch.tab` table were read from to satisfy an SQL delete operation or if it were referenced through an SQL view.
5 INDIVIDUAL TABLES WITH IP ADDRESS WHITE LISTS

This example will use the SPM to create a white list of IP addresses for individual tables. The white list will map a named table to a set of IP addresses from which users are allowed to access the table. Furthermore, the white list will be maintained in a special table and will be updatable using standard SQL operations. Updates to the white list table are restricted to an administrative user from the localhost.

5.1 Database Object Initialization

The database should first be initialized as described in Section 2.2. Next, the spmadmin user should create the policy_data_tab table as follows:

```
spmadmin(U)> rxisql -d db
rxisql:1> create catalog policy_data_cat;
rxisql:2> create schema policy_data_cat.policy_data_sch;
rxisql:3> set schema policy_data_cat.policy_data_sch;
rxisql:4> create table policy_data_tab (table_name varchar(150),
ip_address varchar(50));
rxisql:5> grant exec on catalog policy_data_cat to public;
1 privilege granted.
rxisql:6> grant exec on schema policy_data_cat.policy_data_sch to public;
1 privilege granted.
rxisql:7> grant all privileges on policy_data_cat.policy_data_sch.policy_data_tab to public;
24 privileges granted.
rxisql:8> commit;
rxisql:9> q
spmadmin(U)>
```

Policy data is any RDBMS table data used to make security policy decisions. It should be considered security critical data, much like the shadow passwd file on an operating system. Here we intend to isolate our policy data into its own catalog, schema, and table. This is good practice allowing easy policy control of updates. Note that we have granted DAC privileges to public. This is done because we want to emphasize the behavior of the SPM and not the DAC policy. In practice, it would be better to assign DAC privileges only to administrators of the policy data.

Our policy_data_tab table will contain one row for each allowable table name / IP address mapping. If a row exists in the table then the IP address given by the ip_address column is permitted to access the table given by the table_name column. Note that the values in the ip_address column must adhere to the syntax for the SPM’s ipAddress data type and the values of the table_name column must be the fully specified table names (e.g., db.cat.sch.tab).

5.2 The policy-data Policy Set

The policy-data policy set will control access to the entire policy_data_cat catalog subtree. In practice, controlling the entire subtree with a single policy will make it secure and easy to add more types of table-
stored policy data. In this policy set, we introduce explicitly including a policy within a policy set. The same technique may be used for explicitly including a policy set within another policy set. Note that we re-use the *deny* policy defined in Section 4.2.1. It should be present in the policy repository for this example to function properly. A listing of the *policy-data* policy set follows.
<?xml version="1.0" encoding="UTF-8"?>

<PolicySet

xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

PolicySetId="policy-data"
PolicyScope="Subtree"
PolicyCombiningAlgId="ordered-permitoverrides">

<Description>
Policy for any operation within the policy_data_cat
Allow op only if administrator from localhost
Deny all others
</Description>

<Target/>

<Policy PolicyId="policy-data-admin" RuleCombiningAlgId="ordered-permitoverrides">

<Target>
<Subjects>

<SubjectMatch MatchId="equal">
<AttributeValue DataType="string">spmadmin</AttributeValue>
<SubjectAttributeDesignator AttributeId="subject-name"/>
</SubjectMatch>

<SubjectMatch MatchId="equal">
<AttributeValue DataType="ipAddress">127.0.0.1</AttributeValue>
<SubjectAttributeDesignator AttributeId="ip-address"/>
</SubjectMatch>

</Subjects>

</Target>

<Rule RuleId="permit-all" Effect="Permit"> <Target/> </Rule>

</Policy>

<PolicyIdReference>deny</PolicyIdReference>

</PolicySet>
The policy-data policy set has a scope covering a subtree (line 7). The policy set uses the previously
described ordered-permit-overrides policy combining algorithm. The two enclosed policies will be
evaluated in order and any Permit outcome will take precedence. The policy set contains two policies, the
explicitly included policy-data-admin policy of lines 16-32 and the deny policy which is included by
reference (line 34). The general policy logic is that either the policy-data-admin policy evaluates to
Permit or the deny policy will evaluate to Deny. As before, the deny policy is used as a “catch-all” for an
evaluation environment that does not produce a Permit from any of the other policies.

All logic for the policy-data-admin policy is contained in its target of lines 18-29. The <Subject> element
contains a conjunctive sequence of <SubjectMatch> elements. Therefore, all <SubjectMatch> elements
must evaluate to Match for the <Subject> (and <Target>) to evaluate to Match. In our case, the subject-
name must be spmadmin and the ip-address must be 127.0.0.1 (localhost).

If the target evaluates to Match then the permit-rule rule of line 30 causes the policy (and policy set) to
evaluate to Permit.

If the policy-data-admin policy evaluates to NotApplicable then the deny policy referenced on line 34 will
cause the policy set to evaluate to Deny.

The policy-data policy set as shown in the listing should be placed into a policy configuration files and
added to the repository with the rxpolman –c administrative command. It should then be assigned to the
db.policy_data_cat catalog using the rxpolman –a administrative command.

5.3 The user-table Policy Set

The user-table policy set performs the lookup to determine if the subject’s IP address is approved
according to the rows in the policy_data_tab table. If the IP address is approved for the table being
accessed it permits the operation; otherwise, it denies the operation.

This policy set introduces importing database column values into a policy. Importing database columns
into a policy is the act of extracting all values of any database column whose rows pass a predicate-filter.
The column import process produces results analogous to the SQL statement “select import-column from
import-table where filter-predicate = true.” Internally, all rows in the import table are iterated upon and
the predicate-filter is applied. If a row passes the predicate-filter then the value of the import column is
added to the multi-valued bag. The predicate-filter may reference any column value in the iterated row
using the <ImportFieldSelector> element. The values used for the predicate-filter may come from any
column from the table whose column is being imported.

In our example, the SQL statement that is equivalent to our column import is “select ip_address from
policy_data_tab where table_name = table-name” where table-name is the resource attribute containing
the name of the table currently being operated upon (db.cat.sch.tab). In our policy, the resultant multi-
valued bag will be examined, using the is-in function, to test if the subject attribute ip-address is present.
Presence of the ip-address context attribute in the bag implies the operation is permissible. A listing of
the user-table policy set follows.
<?xml version="1.0" encoding="UTF-8"?>

<PolicySet xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  <PolicySetId>user-table</PolicySetId>
  <PolicyCombiningAlgId>ordered-permit-overrides</PolicyCombiningAlgId>

  <Description>
    Policy for any operation on the user table.
    Allow op only if from allowed IP address as configured in policy_data_tab.
    If a row in policy_data_tab has table_name column equal to the user table name
    then the corresponding value of the ip-address column
    specifies an allowed IP address for the user table.
    Deny all others.
  </Description>

  <Target/>

  <Policy PolicyId="user-table-ip-list" RuleCombiningAlgId="ordered-permit-overrides">
    <Target/>
    <Rule RuleId="ip-list-rule" Effect="Permit">
      <Target/>
      <Condition>
        <Apply FunctionId="is-in">
          <SubjectAttributeDesignator AttributeId="ip-address"/>
          <ImportColumnSelector RequestContextPath="db.policy_data_cat.policy_data_sch.policy_data_tab.ip_address" DataSource="ipAddress">
            <Apply FunctionId="equal">
              <ImportFieldSelector RequestContextPath="db.policy_data_cat.policy_data_sch.policy_data_tab.table_name" DataSource="string"/>
              <ResourceAttributeDesignator AttributeId="table-name"/>
            </Apply>
          </ImportColumnSelector>
        </Apply>
      </Condition>
    </Rule>
  </Policy>

  <PolicyIdReference>deny</PolicyIdReference>

</PolicySet>
As with the policy-data policy set, the user-table policy set contains two policies. The first, user-table-ip-list, will perform the IP address lookup in the policy_data_tab table. The second, deny, will deny all operations not permitted by the first policy. The bulk of the logic for the policy set is contained within the <Condition> element of lines 21-35. The top-level element within the <Condition> is the is-in function (line 22). The is-in function returns true if the first argument, the ip-address <SubjectAttributeDesignator> of line 23, is contained within the multi-valued bag defined by the second argument, the <ImportColumnSelector> of lines 24-33.

The <ImportColumnSelector> element references the policy_data_tab.ip_address column as specified by its RequestContextPath attribute of line 25. The <ImportColumnSelector> contains a single argument (lines 27-32) which defines the predicate-filter. The predicate-filter tests for equality between the value of the policy_data_tab.table_name column (lines 28-30) and the table-name resource attribute (line 31). The value of the policy_data_tab.table_name column is extracted using the <ImportFieldSelector> element. The <ImportFieldSelector> element extracts the table_name column value from the current import table row as the <ImportColumnSelector> loops through all rows in the table.

The user-table policy set as shown in the listing should be placed into a policy configuration file and added to the repository with the rxpolman –c administrative command. It should then be assigned to the db.cat.sch.tab table using the rxpolman –a administrative command.

### 5.4 Policy Behavior

The expected behavior for the tab table is that only a user connecting from a permitted IP address will be able to perform operations. The expected behavior for the policy_data_tab table is that only the spmadmin user connecting from the localhost will be permitted to perform operations. Note that the policy_data_tab table currently contains no rows. We will attempt some operations by the spmuser1 connecting from the localhost on both tables. Note that the IP address has been added to the user’s command prompt.

```
spmuser1(U,127.0.0.1)> rxsql -d db
rxsql:1> select * from cat.sch.tab;
select * from cat.sch.tab
^                  
security policy access violation (RA013)
rxsql:2> select * from policy_data_cat.policy_data_sch.policy_data_tab;
select * from policy_data_cat.policy_data_sch.policy_data_tab
^                  
security policy access violation (RA013)
rxsql:3>q
spmuser1(U,127.0.0.1)> 
```

As expected access to both tables has been denied. We will now add a row in the policy_data_tab permitting access from localhost to the tab table.
As expected the `spmadmin` user is permitted to access the table from `localhost`. Next, we will verify that access to the `cat.sch.tab` table is now permitted for `spmuser1` from `localhost`.

```
spmuser1(U,127.0.0.1) > rxisql -d db
rxsql:1> insert into cat.sch.tab values 'spmadmin', 11, 'NSA';
1 record inserted.
rxsql:2> insert into cat.sch.tab values 'spmuser1', 9, 'CIA';
1 record inserted.
rxsql:3> insert into cat.sch.tab values 'spmuser2', 5, 'DISA';
1 record inserted.
rxsql:4> commit;
rxsql:5> select * from cat.sch.tab;
  tab.name |  tab.pay_grade |  tab.organization
-----------|---------------|------------------
  spmadmin  |     11        |       NSA        
  spmuser1  |       9       |       CIA        
  spmuser2  |       5       |       DISA       
3 records displayed.
rxsql:6> q
spmuser1(U,127.0.0.1) >
```
Next, we will add a second permitted IP address to the `policy_data_tab` table. Note that you must have a remote host configured to connect to the Trusted Rubix database for this part of the example. We will use the IP address 192.168.1.1. You should use the IP address that corresponds to your remote host.

```
spmadmin(U,127.0.0.1)> rxisql -d db
rxsql:1> insert into policy_data_cat.policy_data_sch.policy_data_tab
values 'db.cat.sch.tab', '192.168.1.1';
1 record inserted.
rxsql:2> commit;
rxsql:3> select * from policy_data_cat.policy_data_sch.policy_data_tab;
policy_data_tab.table_name | policy_data_tab.ip_address
---------------------------|------------------------
db.cat.sch.tab             | 127.0.0.1
db.cat.sch.tab             | 192.168.1.1
2 records displayed.
rxsql:4>q
spmadmin(U,127.0.0.1)>
```
Lastly we will verify that spmuser1 may access the tab table from 192.168.1.1.

```
spmuser1(U, 192.168.1.1)> rxisql -d db@dbserver
rxsql:1> select * from cat.sch.tab;
<table>
<thead>
<tr>
<th>tab.name</th>
<th>tab.pay_grade</th>
<th>tab.organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmadmin</td>
<td>11</td>
<td>NSA</td>
</tr>
<tr>
<td>spmuser1</td>
<td>9</td>
<td>CIA</td>
</tr>
<tr>
<td>spmuser2</td>
<td>5</td>
<td>DISA</td>
</tr>
</tbody>
</table>
3 records displayed.
rxsql:2>q
spmuser1(U, 192.168.1.1)>
```
6 ROW ACCESS RESTRICTED TO RDBMS USER CREATOR

The example will use the SPM to control access to rows in a table so that only the RDBMS user who inserted the row will be able to access it. We will refer to the inserter of the row as its creator.

6.1 Database Object Initialization

The database objects should be initialized as described in Section 2.2.

6.2 The user-table-table-ops Policy

The user-table-ops policy allows all table based operations (e.g., table-open). As it is not the intent of this example to control table based operations all are allowed. A listing of the user-table-table-ops policy follows.

```
<xml version="1.0" encoding="UTF-8"?>
<Policy
xmlns="http://www.rubix.com/2008/RXSMLOpen"
xmns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xml:schemaLocation="http://www.rubix.com/2008/RXSMLOpen RXSMLOpen.xsd"
PolicyId="user-table-table-ops"
RuleCombiningAlgId="ordered-permit-overrides"
<Description>
Allow all table operations.
</Description>
<Target>
<Actions>
<Action>
<ActionMatch MatchId="regexp-match">
<AttributeValue DataType="string">table-*</AttributeValue>
<ActionAttributeDesignator AttributeId="action-id"/>
</ActionMatch>
</Action>
</Actions>
</Target>
<Rule RuleId="allow-all" Effect="Permit">
<Target/>
</Rule>
</Policy>
```
The policy simply uses a regular expression match \((\text{regexp-match} \text{ function on line 14})\) to target all operations with a prefix of “table-”. The \textit{allow-all} rule (line 20) permits all operations that match the target.

The \textit{user-table-table-ops} policy as shown in the listing should be placed into a policy configuration file and added to the repository with the \texttt{rxpolman –c} administrative command.

\section*{6.3 The \textit{user-table-insert} Policy}

The \textit{user-table-insert} policy defines security behavior for the row-insert operation. All inserts are permitted and the set-field obligation is used to set the \textit{name} column to the value of the \textit{subject-name} subject attribute. The result will be that all rows in our table will contain their creator in the \textit{name} column. A listing of the \textit{user-table-insert} policy follows.
<?xml version="1.0" encoding="UTF-8"?>

<Policy xmlns="http://www.rubix.com/2008/RXSMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>

    <PolicyId>user-table-insert</PolicyId>
    <RuleCombiningAlgId>ordered-permitoverrides</RuleCombiningAlgId>

    <Description>
        For insert operations:
        Allow all inserts.
        Obligation that sets name column to subject-name of inserter.
    </Description>

    <Target>
        <Actions>
            <Action>
                <ActionMatch MatchId="equal">
                    <AttributeValue DataType="string">row-insert</AttributeValue>
                    <ActionAttributeDesignator AttributeId="action-id"/>
                </ActionMatch>
            </Action>
        </Actions>
    </Target>

    <Rule RuleId="permit-all" Effect="Permit">
        <Target/>
    </Rule>

    <Obligations>
        <Obligation FulfillOn="Permit" ObligationId="set-field">
            <AttributeValue DataType="string">db.cat.sch.tab.name</AttributeValue>
            <SubjectAttributeDesignator AttributeId="subject-name"/>
        </Obligation>
    </Obligations>

</Policy>

The user-table-insert policy target (lines 15-20) matches all row-insert operations. The permit-all rule of line 22 will permit all evaluations that match the target. Therefore, all row-insert operations will be permitted. The obligation of lines 24-27 will use the set-field obligation to set the name column (specified on line 25) with the value of the subject-name context attribute (specified on line 26). Note that the name column will be set prior to the insert taking place in the database and will override any value that was included as part of the SQL insert statement.

The user-table-insert policy as shown in the listing should be placed into a policy configuration file and added to the repository with the rxpolman --c administrative command.
6.4 The *user-table-sel-del-upd* Policy

The *user-table-sel-del-upd* policy controls all row-select, row-delete, and row-update operations on our table. The policy allows all operations that operate on a row created by the subject. Additionally, the policy will deny any update operation that attempts to modify the name column. Denying such updates is necessary to maintain the integrity of the name column always representing the creator. This policy will introduce the `<VariableDefinition>` and `<VariableReference>` elements. These two elements are very useful for writing efficient and modular policy code. The named `<VariableDefinition>` element may be used to represent any expression and the `<VariableReference>` element may be used to reference that expression by name anywhere in the policy. Variable definitions may contain other variable definitions and be nested. A listing of the *user-table-sel-del-upd* policy follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicyId="user-table-sel-del-upd"
RuleCombiningAlgId="ordered-permitoverrides">
  <Description>
    Permit select, delete, and update operations only by row creator.
    Do not permit name column to be updated.
  </Description>
  <Target/>

  <VariableDefinition VariableId="is-creator">
    <Apply FunctionId="equal">
      <SubjectAttributeDesignator AttributeId="subject-name"/>
      <AttributeSelector RequestContextPath="db.cat.sch.tab.name" DataType="string"/>
    </Apply>
  </VariableDefinition>
</Policy>
```
Lines 13-17 contain a variable definition expression that returns true if the current subject is the creator of the current row; otherwise, it returns false. A variable definition may contain any expression that evaluates to a single or multi-valued bag. It may be referenced by name multiple times within its enclosed policy. In our policy it is referenced on lines 33 and 44.

Our policy contains two rules which are combined with the ordered-permit-overrides rule combining algorithm. The first rule, select-delete-rule (lines 19-34) controls all row-select and row-delete operations on our table. The target of lines 22-31 matches all such operations. The rule’s condition (line 33) consists of a reference to the is-creator variable definition of lines 13-17. The logic of the select-delete-rule rule is
to allow only select and delete operations on rows by its creator. The second rule, update-rule (lines 35-49), controls all row-update operations on our table. It is similar to the select-delete-rule with the added requirement that the name column not be updated. Its condition uses and logic to combine the is-creator variable reference (line 44) with a check of whether the name column is part of the update operation’s column-name resource attribute (lines 45-49).

The user-table-sel-del-upd policy as shown in the listing should be placed into a policy configuration file and added to the repository with the rxpolman –c administrative command.

6.5 The user-table Policy Set

The user-table policy set contains all of the policy logic for the db.cat.sch.tab table. It simply references each policy previously discussed. It evaluates to permit if any policy evaluates to permit. Note that this policy uses the deny policy described in Section 4.2.1. A listing of the user-table policy set follows.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <PolicySet
3    xmlns="http://www.rubix.com/2008/RXSMLSchema"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://www.rubix.com/2008/RXSMLSchema RXSMLSchema.xsd"
6    PolicySetId="user-table"
7    PolicyCombiningAlgId="ordered-permit-overrides">
8  <Description>
9    Logic for user table object
10   First: permit if table-* operation
11   Second: permit select and delete if performed by owner (subject-name equals name column)
12   and permit update if by creator and not updating name column.
13   Third: permit if insert, but force name column to be subject-name attribute
14   Fourth: deny all others
15  </Description>
16  <Target/>
17  <PolicyIdReference>user-table-table-ops</PolicyIdReference>
18  <PolicyIdReference>user-table-sel-del-upd</PolicyIdReference>
19  <PolicyIdReference>user-table-insert</PolicyIdReference>
20  <PolicyIdReference>deny</PolicyIdReference>
21  </PolicySet>
```
The user-table policy set as shown in the listing should be placed into a policy configuration file and added to the repository with the rxpolman –c administrative command. It should then be assigned to the db.cat.sch.tab table using the rxpolman –a administrative command.

### 6.6 Policy Behavior

Our expected behavior is that users may only access rows which they created. Also, we expect update operations to fail if they attempt to update the `name` column.

First, we will insert a row using the `spmadmin` user as follows.

```plaintext
spmadmin(U)> rxisql -d db
rxsql:1> insert into cat.sch.tab values 'whatever', 11, 'NSA';
1 record inserted.
rxsql:2> commit;
rxsql:3> select * from cat.sch.tab;
tab.name | tab.pay_grade | tab.organization
------- | -------------- | ------------------
       |     11         |     NSA
1 record displayed.
rxsql:4>q

Note that the name column value correctly represents the creator of the row, `spmadmin`, despite the insert value of "whatever". Next, we will do the same with the `spmuser1` user.

```plaintext
spmuser1(U)> rxisql -d db
rxsql:1> insert into cat.sch.tab values 'foobar', 9, 'CIA';
1 record inserted.
rxsql:2> commit;
rxsql:3> select * from cat.sch.tab;
tab.name | tab.pay_grade | tab.organization
------- | -------------- | ------------------
       |        9      |      CIA
1 record displayed.
rxsql:4>q
```

Again, note that despite the values used in the insert statement, the value of the `name` column is properly set to the creator. Also note that the select statement only received a single row despite the table having two rows. This is our expected security behavior where only the creator may select a row and all other rows will be filtered.
Finally, we see similar and proper behavior for the spmuser2 user. Our table now has the three usual rows in it. Next, we will demonstrate the update and delete behavior.

```
spmadmin(U)> rxisql -d db
rxsql:1> update cat.sch.tab set name = 'foobar' where name = 'spmadmin';
update cat.sch.tab set name = 'foobar' where name = 'spmadmin'
^  
security policy access violation (RA013)
rxsql:2> update cat.sch.tab set pay_grade = 12 where name = 'spmadmin';
1 record updated.
rxsql:3> commit;
rxsql:4> select * from cat.sch.tab;
tab.name | tab.pay_grade | tab.organization
----------- | --------------- | ---------------
spmadmin   | 12              | NSA
1 record displayed.
rxsql:5> update cat.sch.tab set pay_grade = 99 where name = 'spmuser1';
There were no records updated.
rxsql:6> delete from cat.sch.tab;
1 record deleted.
rxsql:7> select * from cat.sch.tab;
There were no records selected.
rxsql:8>q
spmadmin(U)>
```

Our first update operation is denied because we are trying to update the name column, which would violate the condition that the name column contains its creator. The second update operation succeeds because we are updating a row created by spmadmin and we are not updating the name column. The update is verified by the next select operation. The next update, where spmadmin explicitly tries to update a row created by spmuser1 the operation updates no rows. This is because the row created by spmuser1 is filtered from the result set when spmadmin reads from the table. Thus, when the SQL operation “searches” for rows where name = “spmuser1” none are found. This behavior can be modified so that this type of update would return a policy violation error using the set-error-code obligation. Next, we attempt to delete all rows in the table. Only one row is deleted, the row that spmadmin created.
7 ROW ACCESS RESTRICTED TO APPLICATION USER CREATOR

This example will use the SPM to control access to rows in a table so that only the user who inserted the row will be able to access it. We will refer to the inserter of the row as its creator.

This example will simulate a simple internet banking RDBMS application. Typically, the RDBMS application is a stand-alone program that is created using the Trusted Rubix ODBC client libraries. The RDBMS application program is executed by a RDBMS user that may connect to the database. The RDBMS application accepts operations from application users that connect to the application through the Internet. The application maps the application users’ operations (e.g., read my bank balance) into SQL operations (e.g., select the balance from the account table) and submits them to the database. For our simulated example, we will use the Trusted Rubix rxisql client to submit SQL operations that correspond to typical operations the RDBMS application would perform.

7.1 Database Object Initialization

The creation of all database objects should be performed by the spmadmin user. Note that the operating system shell prompt of all examples displays the user performing the operation and the current session label (e.g., spmadmin(U)>). The Trusted Rubix RXISQL client will always have a prompt similar to rxsql:1>.

First ensure the db database does not exist and then create the db database with the following commands:

```plaintext
spmadmin(U)> rxdb -df db
spmadmin(U)> rxisql -d master
rxsql:1> create database db;
rxsql:2> q
spmadmin(U)>
```
Next we create the `cat` catalog, `sch` schema, and `tab` table. To simplify the tutorial, we will give required DAC permissions for all objects to `public`.

```
spmadmin(U)> rxisql -d db
rxsql:1> create catalog cat;
rxsql:2> create schema cat.sch;
rxsql:3> create table cat.sch.tab (app_userid integer, account_num integer, account_balance numeric(15,2) default 0.00);
rxsql:4> grant exec on database to public;
1 privilege granted.
rxsql:5> grant exec on catalog cat to public;
1 privilege granted.
rxsql:6> grant exec on schema cat.sch to public;
1 privilege granted.
rxsql:7> grant all privileges on table cat.sch.tab to public;
32 privileges granted.
rxsql:8> create application myapp;
rxsql:9> create application_admin application = myapp user = spmuser1;
rxsql:10> commit;
rxsql:11> q
spmadmin(U>)
```

The `cat.sch.tab` table represents bank account information for a simplistic internet banking application. The table contains three columns `app_userid`, `account_num`, and `account_balance`. Note that the hidden column `rowlabel` also exists. The `app_userid` column will contain the user ID of the application user that inserts a row. It will be automatically set using the set-field obligation in the `user-table-insert` policy. The `account_number` column will hold the account number of the user’s bank account while the `account_balance` column will hold the account’s current balance.

In `rxsql` step 8 an application named `myapp` is created within the database. In `rxsql` step 9 the RDBMS user `spmuser1` is assigned as an application administrator for the `myapp` application. This will allow `spmuser1` to set `myapp` as the current application, create applications users for the `myapp` application, and submit SQL operations on behalf of application users.

Now that `spmuser1` has been assigned as an application administrator for the `myapp` application, application users will be created as follows.

```
spmuser1(U)> rxisql -d db
rxsql:1> alter session set application = myapp;
rxsql:2> create application_user appuser1 with password 'password1';
rxsql:3> create application_user appuser2 with password 'password2';
rxsql:10> commit;
rxsql:11> q
spmuser1(U>)
```

These SQL operations simulate the RDBMS application initializing as it must set the current application to `myapp` before it can perform any application related operations. It then creates two application users,
appuser1 and appuser2. These application users may subsequently be authenticated to the database only while the myapp application is set as current.

7.2 The user-table-table-ops Policy

The user-table-ops policy allows all table based operations (e.g., table-open). As it is not the intent of this example to control table based operations all are allowed. A listing of the user-table-table-ops policy follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Policy
    xmlns="http://www.rubix.com/2008/RXSMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    PolicyId="user-table-table-ops"
    RuleCombiningAlgId="ordered-permit-overrides">
  <Description>
    Allow all table operations.
  </Description>
  <Target>
    <Actions>
      <Action>
        <ActionMatch MatchId="regexp-match">
          <AttributeValue DataType="string">table-*</AttributeValue>
          <ActionAttributeDesignator AttributeId="action-id"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
  <Rule RuleId="allow-all" Effect="Permit">
    <Target/>
  </Rule>
</Policy>
```

The policy simply uses a regular expression match (regexp-match function on line 14) to target all operations with a prefix of “table-“. The allow-all rule (line 20) permits all operations that match the target.

The user-table-table-ops policy as shown in the listing should be placed into a policy configuration file and added to the repository with the rxpolman –c administrative command.
7.3 The *user-table-insert* Policy

The *user-table-insert* policy defines security behavior for the row-insert operation. All inserts are permitted and the set-field obligation is used to set the *app_userid* column to the value of the *application-user-id* subject attribute. The *application-user-id* subject attribute uniquely identifies an application user within an entire database while the *application-user-name* subject attribute uniquely identifies an application user only within the application. Therefore, if *application-user-name* is used for security enforcement it should be used in conjunction with the *application-name* subject attribute.

The result of the policy execution will be that the *app_userid* column will contain the ID of the application user that created each row. This value will be used by other policies to control access to the row. Note that if no application user is currently authenticated, the insert operation will be refused. A listing of the *user-table-insert* policy follows.
<?xml version="1.0" encoding="UTF-8"?>

<Policy xmlns="http://www.rubix.com/2008/RXSMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>
    PolicyId="user-table-insert"
    RuleCombiningAlgId="ordered-permit-overrides">

    <Description>
        For insert operations:
        Allow all inserts.
        Obligation sets app_userid column to application-user-id of inserter.
    </Description>

    <Target>
        <Actions>
            <Action>
                <ActionMatch MatchId="equal">
                    <AttributeValue DataType="string">row-insert</AttributeValue>
                    <ActionAttributeDesignator AttributeId="action-id"/>
                </ActionMatch>
            </Action>
        </Actions>
    </Target>

    <Rule RuleId="permit-all" Effect="Permit">
        <Target/>
    </Rule>

    <Obligations>
        <Obligation FulfillOn="Permit" ObligationId="set-field">
            <AttributeValue DataType="string">
                db.cat.sch.tab.app_userid
            </AttributeValue>
            <SubjectAttributeDesignator AttributeId="application-user-id"
                MustBePresent="true"/>
        </Obligation>
    </Obligations>

</Policy>

The user-table-insert policy target (lines 15-20) matches all row-insert operations. The permit-all rule of line 22 will permit all evaluations that match the target. Therefore, all row-insert operations will be permitted. The obligation of lines 24-28 will use the set-field obligation to set the app_userid column (specified on line 26) with the value of the application-user-id context attribute (specified on line 27). Note that the app_userid column will be set prior to the insert taking place in the database and will override any value that was included as part of the SQL insert statement.
The user-table-insert policy as shown in the listing should be placed into a policy configuration file and added to the repository with the `rxpolman -c` administrative command.

### 7.4 The `user-table-sel-del-upd` Policy

The `user-table-sel-del-upd` policy controls all row-select, row-delete, and row-update operations on our table. The policy allows all operations that operate on a row created by the subject. Additionally, the policy will deny any update operation that attempts to modify the `app_userid` column. Denying such updates is necessary to maintain the integrity of the `app_userid` column always representing the creator. This policy will use the `<VariableDefinition>` and `<VariableReference>` elements. These two elements are very useful for writing efficient and modular policy code. The named `<VariableDefinition>` element may be used to represent any expression and the `<VariableReference>` element may be used to reference that expression by name anywhere in the policy. Variable definitions may contain other variable definitions and be nested. A listing of the `user-table-sel-del-upd` policy follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicyId="user-table-sel-del-upd"
RuleCombiningAlgId="ordered-permit-overrides">
<Description>
  Permit select, delete, and update operations only by app user who created row.
  Do not permit app_userid column to be updated.
</Description>
<Target/>

<VariableDefinition VariableId="is-creator">
  <Apply FunctionId="equal">
    <SubjectAttributeDesignator AttributeId="application-user-id"/>
    <AttributeSelector RequestContextPath="db.cat.sch.tab.app_userid" DataType="integer"/>
  </Apply>
</VariableDefinition>
```

© 2014 Infosystems Technology, Inc.
Lines 13-17 contain a variable definition expression that returns \textit{true} if the current subject is the creator of the current row; otherwise, it returns \textit{false}. A variable definition may contain any expression that evaluates to a single or multi-valued bag. It may be referenced by name multiple times within its enclosed policy. In our policy it is referenced on lines 32 and 44.

Our policy contains two rules which are combined with the \textit{ordered-permit-overrides} rule combining algorithm. The first rule, \textit{select-delete-rule} (lines 19-34) controls all \textit{row-select} and \textit{row-delete} operations on our table. The target of lines 22-31 matches all such operations. The rule’s condition (line 33) consists of a reference to the \textit{is-creator} variable definition of lines 13-17. The logic of the \textit{select-delete-rule} rule is to allow only select and delete operations on rows by its creator. The second rule, \textit{update-rule} (lines 35-
49), controls all row-update operations on our table. It is similar to the select-delete-rule with the added requirement that the name column not be updated. Its condition uses and logic to combine the is-creator variable reference (line 44) with a check of whether the name column is part of the update operation’s column-name resource attribute (lines 45-48).

The user-table-set-del-upd policy as shown in the listing should be placed into a policy configuration file and added to the repository with the rxpolman –c administrative command.

### 7.5 The user-table Policy Set

The user-table policy set contains all of the policy logic for the db.cat.sch.tab table. It simply references each policy previously discussed. It evaluates to permit if any policy evaluates to permit. Note that this policy uses the deny policy described in Section 4.2.1. A listing of the user-table policy set follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet
xmlns="http://www.rubix.com/2008/RXSMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
PolicySetId="user-table"
PolicyCombiningAlgId="ordered-permit-overrides">
   <Description>
      Logic for user table object
      First: permit if table-* operation
      Second: permit select and delete if performed by app user owner
      and permit update if by creator and not updating name column.
      Third: permit if insert, but force app_userid column to be application-user-id attribute
      Fourth: deny all others
   </Description>
   <Target/>
   <PolicyIdReference>user-table-table-ops</PolicyIdReference>
   <PolicyIdReference>user-table-set-del-upd</PolicyIdReference>
   <PolicyIdReference>user-table-insert</PolicyIdReference>
   <PolicyIdReference>deny</PolicyIdReference>
</PolicySet>
```

The user-table policy set as shown in the listing should be placed into a policy configuration file and added to the repository with the rxpolman –c administrative command. It should then be assigned to the db.cat.sch.tab table using the rxpolman –a administrative command.
7.6 Policy Behavior

Our expected behavior is that a row may be accessed only while the application user that inserted the row is currently authenticated. Also, we expect update operations to fail if they attempt to update the app_userid column.

**Note that the authentication of an application user is subject to a timeout.** The timeout defaults to 10 minutes and is settable in the rxconfig configuration settings file. Therefore, if the authentication reaches the timeout, the behavior will change from that expected in the example.

First, we will attempt to insert a row using the spmuser1 user as follows:

```
spmuser1(U)> rxisql -d db
rxisql:1> alter session set application = myapp;
rxisql:2> insert into cat.sch.tab values 99999, 1, 0.0;
insert into cat.sch.tab values 99999, 1, 0.0
    ^
security policy access violation (RA013)
rxisql:3>q
spmuser1(U>)
```

Note that no application user is authenticated. The insert operation fails because the application-user-id subject attribute is required for the set-field obligation of the policy and it does not exist. It only exists in the session while an application user is authenticated. The denial of the insert operation ensures that a row is never created with an invalid value for the app_userid column.

Next we will insert a row while there is an authenticated application user.

```
spmuser1(U)> rxisql -d db
rxisql:1> alter session set application = myapp;
rxisql:2> authenticate application_user = appuser1 password = 'password1';
rxisql:3> insert into cat.sch.tab values 99999, 1, 0.0;
1 record inserted.
rxisql:4> select * from cat.sch.tab;
tab. app_userid | tab.account_num | tab.account_balance
-----------------|-----------------|---------------------
                |                 |                     |
| 1              | 1               | 0.0                 |
1 record displayed.
rxisql:5>commit;
rxisql:6>q
spmuser1(U>)
```

In this case the insert was successful because we have an authenticated application user, appuser1. Note that the app_userid column value correctly represents the ID of the application user (1) that created the row despite the insert value of “99999”. Next, we will do the same with the appuser2 application user.
Again, note that despite the values used in the insert statement, the value of the app_userid column is properly set to the creator. Also note that the select statement only received a single row despite the table having two rows. This is our expected security behavior where only the application user that created the row may select a row and all other rows will be filtered.

Finally, we see the behavior when selecting from the table with no application user authenticated. All rows are filtered from view. Thus, even if a malicious entity has complete control of the RDBMS application, they will only be able to see rows for application users which are currently authenticated. If not such users are authenticated then they will see no rows. Next, we will demonstrate the update and delete behavior.
Our first update operation is denied because we are trying to update the `app_userid` column, which would violate the condition that the `app_userid` column contains its creator.

The second update operation succeeds because we are updating a row created by `appuser1` and we are not updating the `app_userid` column. The update is verified by the next select operation. Note that despite attempting to update all rows due to the SQL statement having no WHERE clause, only the row created by `appuser1` was updated.

The next update, where `appuser1` explicitly tries to update a row created by `appuser2` the operation updates no rows. This is because the row created by `appuser2` is filtered from the result set when `appuser1` reads from the table. Thus, when the SQL operation “searches” for rows where `app_userid` = 2, none are found. This behavior can be modified so that this type of update would return a policy violation error using the `set-error-code` obligation. Next, we attempt to delete all rows in the table. Only one row is deleted, the row that `appuser1` created.