Trusted RUBIX™
Version 6

SELinux Guide

Revision 8

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

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.

Printed in U.S.A.

Introduction

This document provides an overview of the SELinux security mechanism as it relates to the Trusted RUBIX Relational Database Management System (RDBMS). It also provides a guide to using the Trusted RUBIX SELinux policy to create custom RDBMS policies.

This guide is **not meant to be an exhaustive reference** for SELinux. Many of the SELinux policy language rules are not included in this document. For more information on SELinux please consult the following resources.

The book *SELinux by Example* (ISBN: 0-13-196369-4) provides a good, but somewhat outdated, introduction to using and configuring SELinux.

For more information about the SELinux policy on RHEL6 see the following URL:

https://access.redhat.com/knowledge/docs/Red_Hat_Enterprise_Linux/

For more recent information about SELinux see the SELinux Notebook at the following URL:

http://www.freetechbooks.com/the-selinux-notebook-the-foundations-t785.html

For more information about the SELinux policy on Fedora see the following URL:

http://docs.fedoraproject.org/selinux-user-guide/f13/en-US/

General information about SELinux, including an active mailing list, may be found at:

http://www.nsa.gov/research/selinux/

Active mail lists exist where specific questions may be answered. Information about the general SELinux mailing list may be found at:

http://www.nsa.gov/research/selinux/list.shtml

SELinux Overview

SELinux is a security policy enforcement mechanism integrated into the Linux operating system. It is based upon the Flask security model. The SELinux security model assigns every Linux object (file, directory, socket, process, etc.) an *object class* and a set of operations, also called *permissions*, on the object class. The model assigns a *type* to each instantiated subject and object. The type assigned to each instantiated subject (e.g., process) is generally referred to as a *domain* or a *domain type*. The heart of the SELinux mechanism is a set of rules that define which operations a subject with a specific domain may perform given the target object's class and type. The enforcement of these rules is known as *Type Enforcement* (TE). The sum of these rules basically constitutes a large access control list (ACL) for object class operations, domains, and object types. Each element in the ACL would indicate, with a permit or deny, if a subject with the given domain is able to perform the operation upon an object with the given object class and type. SELinux denies an operation unless there is a specific TE rule permitting it. In addition to TE rules that allow or deny an operation, rules also exist

that determine how types are assigned to subjects and objects (i.e., how subjects and objects are labeled).

Each subject and object is assigned a string based security label called a *context*. The context consists of four components: the SELinux user, the role, the type (or domain), and a Multi-level Security/Multi-Category Security (MLS/MCS) level range. An object's context is calculated and assigned when the object is created and is generally static. A subject's context typically changes during the subject's session. All components except the SELinux user may change. Changes to a subject's context occur through the execution of programs that are configured to cause a context transition (when a specially configured program is executed) or through an explicit user command (e.g., the *newrole* or *sudo* command).

The *SELinux user* is assigned during login and may not change during the session It is distinct from and should not be confused with the traditional Linux login user. Linux login users are mapped to a single SELinux user while each SELinux user may be associated with multiple Linux login users. Each SELinux user has an associated set of roles and a subject may assume a role only if the role is associated with its SELinux user. Therefore, the SELinux user bounds the potential set of roles a subject may assume.

At any given time a subject is assigned a single *role*. Each role has an associated set of types and a subject may transition to a type only if the type is associated with its role. Furthermore, a subject may only transition from one role to another role if rules are defined allowing the transition. Therefore, a subject's role, along with the SELinux policy rules, defines which type and role transitions may occur. A subject's initial role is either explicitly set upon login or is taken from the assigned default role.

In addition to TE rule enforcement, SELinux also optionally enforces multi-level security (MLS) or multi-category security (MCS). MLS enforce a Bell-Lapadula policy over subjects and objects. MCS is similar to MLS in that all levels are restricted to a single sensitivity level with multiple categories. Additionally, MCS is a discretionary policy allowing an object owner to set the categories for that object[1]. For an operation to be permitted the TE rules and the MLS/MCS rules must both be satisfied. A bounding level-range is defined for each SELinux user.

The SELinux policy rules are specified as a text based scripting language and are compiled before being installed to a system. Type Enforcement rules, SELinux users, roles, types, and MLS rules may all be defined within the policy. As policy behavior generally is defined for all subjects and objects in a system, the policy code base may be large. To simplify policy development, *interfaces* (analogous to procedure calls) are used to provide a more programmatic environment for creating complex policies. The open source *Reference Policy*, which serves as a basis for developing policies for specific operating systems, and several open source policy development tools are maintained by Tresys Technology. The SELinux base policy and policy modules for specific applications are available as installable packages and as source code. Custom security policy modules may be written and installed by the security administrator, including policy modules that customize the security behavior of the Trusted RUBIX RDBMS.

The Trusted RUBIX RDBMS is SELinux security enforcing software. It interacts with the SELinux functionality of the underlying operating system to extend SELinux security controls to all Trusted

[1] Trusted RUBIX does not support the discretionary aspects of the MCS policy; that is, an owner of a Trusted RUBIX RDBMS object may not change the MCS categories associated with that object.

RUBIX RDBMS subjects and objects. SELinux policy rules may be added to the operating system's SELinux policy repository that will define the security behavior for the RDBMS subjects and objects. Custom SELinux roles and security behavior may be created that will define the security behavior over both operating system and RDBMS objects, allowing for a coherent security policy across all objects on the platform.

SELinux Context

Each subject and object in the system is assigned an SELinux context (security label). The context is string based and consists of four parts: the SELinux user, the role, the type (or domain), and the MLS/MCS level range. The following is an example of an SELinux context:

```
user_u:user_r:user_t:s0-s15:c0.c1023
```

The first component, *user_u*, is the SELinux user. The SELinux user is distinct from and should not be confused with the traditional Linux login user. It is assigned during login and may not change during the session. Linux login users are mapped to a single SELinux user and assume that SELinux user at login. SELinux users have a set of permitted roles associated with them. For objects, the SELinux user component of its context generally represents the SELinux user that created the object.

The second component, *user_r*, is the SELinux role. The subject's current role may change during a login session, generally by an explicit command (e.g., *newrole*). The current role defines which domain types may be assumed and to which roles it may transition. If a type is not valid for a given role, the corresponding context is considered invalid and no object or subject may be assigned that context. For objects, the role is generally set to *object_r* and has no special significance.

The third component, *user_t*, is the SELinux type. The subject's current type, along with the SELinux Type Enforcement rules, define which object class permissions it has. A subject's type is assigned at login and may change either through an explicit command or while executing programs through TE type transition rules. The object's type, the subject's domain type, and the TE rules, control which operations a subject may perform. An object's type is generally calculated during its creation based upon the creating subject's type and the parent object's type. File system objects may be explicitly typed using *regex* rules written upon the object's path.

The last component, *s0-s15:c0.c1023*, is the SELinux MLS/MCS level range. The level range bounds the operations of the subject. For our example the low end of the range is *s0* and the high end of the range is *s15:c0.c1023*. The low end of the range is the subject's current session level and the high end of the range represents its clearance. The example level given is in its raw format. In this format the sensitivity component of the level is given by the '*s*' component where *s0* < *s1* < *s2* etc (the '<' symbol represents a "strictly dominated by" relationship) . The categories are given by the '*c*' component. The value *c0.c1023* represents all categories between category c0 and category c1023 inclusive (the '.' represents an inclusive range of categories) while *c1,c*5 would represent category c1 and category c5 (the comma represents a list of categories). In addition to raw level formats, if the *mcstrans* level translation service is installed and enabled, user friendly level names may be used such as *SystemHigh*. These may be configured using the SELinux Management tool. While the SELinux MLS functionality does not require a Bell-Lapadula policy, this is currently the only SELinux MLS functionality available and the only one compatible with the Trusted RUBIX RDBMS.

It is common practice, but not a requirement, to use the _u, _r, and _t suffix for SELinux users, roles,

and types respectively.

An important concept regarding the SELinux context is its validity. In order for a context to be valid the following must be true:

- → The SELinux user must exist in the system
- → The role must be associated with the SELinux user
- → The type must be associated with the role
- → The MLS level range must contain valid levels for the system and the SELinux user

A new context is created when an object is labeled and when a domain context transition occurs. Often, the security is enforced simply by validating the resultant context and rejecting the operation if the context is invalid. For instance, when a role transition occurs (e.g., when the *newrole* command is invoked) the new role must be associated with the SELinux user. Additionally, when a domain type transition occurs (e.g., when a program is executed with associated domain transition rules) the new domain type must be valid for the given role.

Object Classes and Permissions

Every SELinux-protected object in the operating system has an associated object class and each object class has an associated permissions vector. The permissions vector defines the operations that are performed on the object and are used in TE rules to control access. An example is the *file* object class. The *file* object class has a permissions vector that includes *create*, *append*, *getattr*, *read*, *write*, and others. The set of object classes may be found in the */usr/include/selinux/flask.h* header file. A list of permissions may be found in */usr/include/selinux/av_permissions.h* header file. Object classes and permissions are generally fixed and may not be defined in custom policy modules.

There are a set of RDBMS object classes (e.g., *db_table*) and permissions (e.g., *select*, *insert*) that are used by Trusted RUBIX to control access to RDBMS objects. RDBMS object classes and permissions used by Trusted RUBIX are enumerated later in this document.

Declaring SELinux Roles and Types

Before using a specific role or type it must first be declared in the policy.

A role declaration is performed using the *role* declaration statement. The syntax for the *role* statement follows:

```
role role_name [types type_set];
```

The following example shows the declaration of a role named *rubix_user_r*:

```
role rubix_user_r;
```

The *role* statement is also used to define the set of types that are valid for the role. If a type is not valid for a role then it may not co-exist with the role in any context. In such cases the context creation will fail as will the associated operation. The set of types may be included on the initial role declaration or in any number of subsequent *role* rules. An example role declaration that associates the *rubix_user_t* and the *user_t* types with the *rubix_user_r* role follows. If the *rubix_user_r* had not been

previously declared then this rule would also declare it.

```
role rubix_user_r types rubix_user_t, user_t;
```

A type declaration is performed using the *type* declaration statement. The syntax for the *type* statement follows:

```
type type_name [alias alias_set] [, attribute_set];
```

The following example shows the declaration of a type named *rubix_user_t*:

```
type rubix_user_t;
```

An *alias* set for the type name may be optionally assigned in the *type* statement. An alias may be used in lieu of the type name in subsequent rules and statements. There is a one-to-one relationship between an alias and a type name. Attributes may also be associated with a type using the type statement as discussed later in this document. There is a many-to-many relationship between attributes and type names.

An alias may be declared for a pre-defined type using the *typealias* statement as follows:

```
typealias rubix_user_t dbms_user;
```

Role Transitions

An SELinux user represents a bounded set of roles a subject may assume. An SELinux role represents a bounded set of types that a subject may “reach." When a Linux user initiates a session it is assigned an SELinux user according a to one-to-one mapping in the policy. The assigned SELinux user will have a set of roles associated with it, one being a default role. The default role is initially assigned to the subject. Other roles in the SELinux user’s role set may be assumed through a role transition.

In order for a role transition to be permitted a corresponding role *allow* rule must exist in the policy. The syntax for the role allow rule is:

```
allow source_role_set target_role_set;
```

If multiple entries are specified in a role set they must be enclosed in braces ({}). The role *allow* rule permits every role in the source role set to transition to a role in the target role set. An example role *allow* rule is:

```
allow rubix_staff_r {rubix_dbadmin_r rubix_secadmin_r};
```

This rule permits the subjects in the *rubix_staff_r* role to transition to the *rubix_dbadmin_r* and *rubix_secadmin_r* roles.

USING THE *newrole* COMMAND

For the purposes of using Trusted RUBIX, role transitions generally occur through the use of the *newrole* command.

Roles may be assumed by logging in as a user with the desired role configured to be the user’s default

role or by using the *newrole* command. If the *newrole* command is not available you may need to install the *policycoreutils-newrole* package.

To use the *newrole* command to assume a new role there must be SELinux policy rules that allow transition from the source role to the target role. By default, Trusted RUBIX allows the administrative and client roles to be reached from all Linux login roles (e.g., *unconfined_r, user_r*) roles. Therefore, Trusted RUBIX roles may be distributed, according to security requirements, using SELinux user definitions and their mappings to Linux login users. Note that the *unconfined_r* is only available while using the Targeted policy.

As an example, to reach the *rubix_dbadm_r* role first become the *user_r* or *unconfined_r* role. This is usually accomplished by having the *user_r* or *unconfined_r* be the default role of a login user. Then the *newrole* command is used to assume the new role as follows:

```
newrole -r rubix_dbadm_r
```

Additionally, the *newrole* command may be used to change the current type or MLS/MCS level.

Role transition rules may be given through using the SELinux base policy interface as follows:

userdom_role_change_template(source_role_prefix, dest_role_prefix)

See the */var/lib/RUBIXdbms/etc/selinux/rubix-dev.te* policy source file for examples of its use.

RDBMS ROLE TRANSITION EXAMPLE

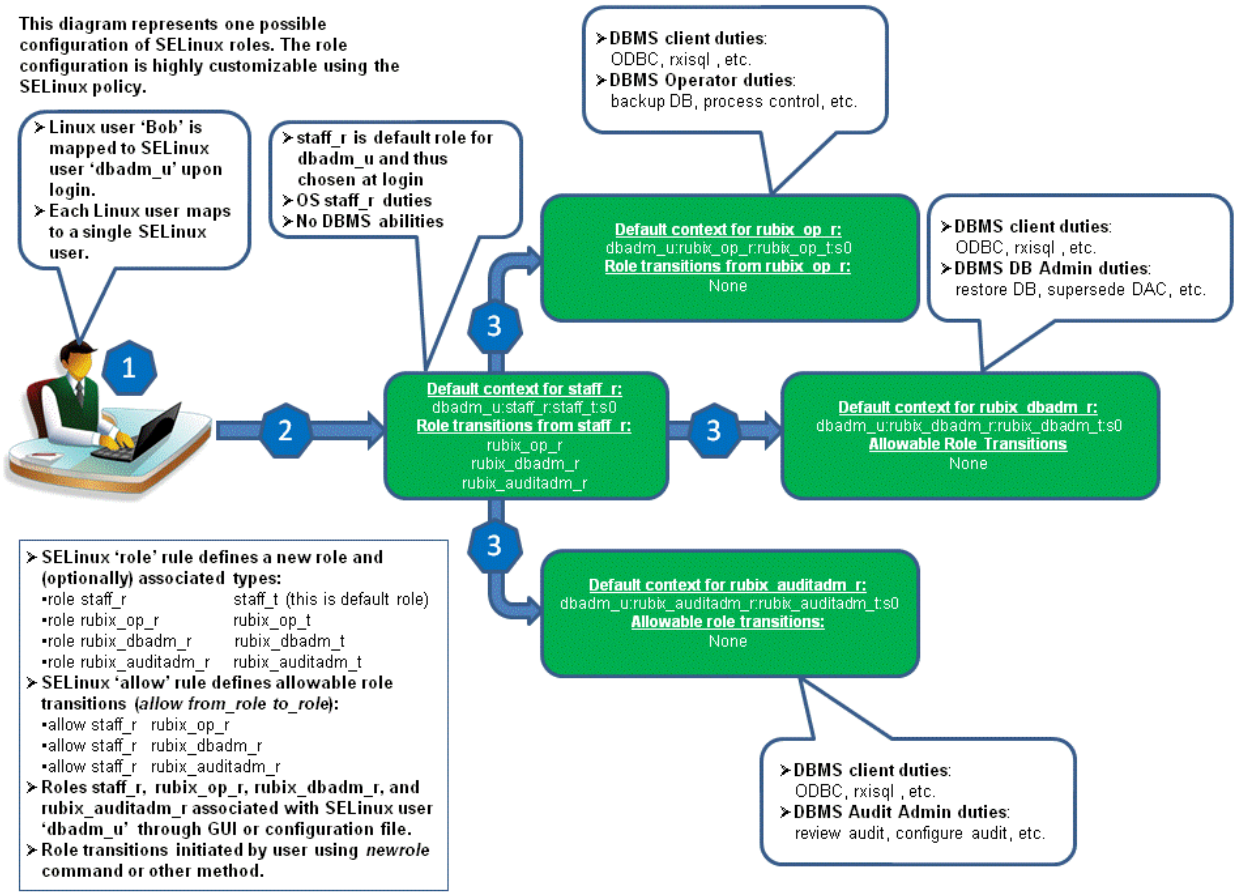
The following diagram illustrates a user logging into the system and transitioning to a role that is able to execute Trusted RUBIX operations. The text box in the lower left corner contains actual SELinux policy rules that correspond to the diagram.

Step 1: Linux user *Bob* logs into the operating system. During login, the Linux user *Bob* is mapped to the SELinux user *dbadm_u*.

Step 2: An initial SELinux context is automatically constructed for user *Bob*. The SELinux user component is from Step 1, *dbadm_u*; the role component is the default role for the *dbadm_u* SELinux user, *staff_r*; the type component is the default type for the *staff_r* role, *staff_t*; the MLS/MCS level range is taken from the *dbadm_u* SELinux user configuration and is *s0-s0*. Note that if the high and low level ranges are the same, they are displayed as a single value (i.e., *s0*). The SELinux context for Bob upon login is *dbadm_u:staff_r:staff_t:s0*. Typically, the *staff_r* role would be configured to allow some operating system administrative abilities, but no Trusted RUBIX abilities. The user must transition into a new role to be able to perform RDBMS duties.

Step 3: The user *Bob* explicitly transitions to one of the reachable Trusted RUBIX roles: *rubix_op_r, rubix_dbadm_r,* or *rubix_auditadm_r*. According to the SELinux policy rules, these are the only three roles that may be reached from the *staff_r* role. In our example, the user would use the operating system *newrole* command to explicitly transition to a role. Once the user transitioned to that role, he would be able to perform Trusted RUBIX RDBMS duties associated with that particular role, such as executing an ODBC application or performing an administrative operation (e.g., backup the database).

Figure 1: Login and Role Transition



Type Enforcement Rules

The basic TE rule is the *allow* rule. The *allow* rule permits operations on objects of a given class set and of types of a given type set (*target_type_set*) by subjects of a given domain set (*source_type_set*). The syntax of the *allow* rule is:

```
allow source_type_set target_type_set : class_set perm_set ;
```

If a set has multiple entries it must be enclosed in braces ({ }). A type set may contain types or attributes. A type may be excluded from the set by preceding it with a '-' (useful only if attributes are used). All permissions in the *perm_set* must be valid for all object classes in the *class_set*.

An example *allow* rule is:

```
allow rubix_user_t rubix_table_t : db_table { select insert };
```

This rules allows subjects with the *rubix_user_t* type to perform *select* and *insert* operations on objects of the *db_table* object class with a type of *rubix_table_t*.

To prevent an operation ever from being permitted, despite corresponding *allow* rules, the *neverallow*

rule may be used. The *neverallow* rule always takes precedence over an *allow* rule. The *neverallow* rule denies operations on objects of a given class set and of types of a given type set (*target_type_set*) by subjects of a given domain set (*source_type_set*). The syntax of the *neverallow* rule is:

```
neverallow source_type_set target_type_set : class_set perm_set ;
```

If a set has multiple entries it must be enclosed in braces ({ }). A type set may contain types or attributes. A type may be excluded from the set by preceding it with a '-' (useful only if attributes are used). A type set may also be specified as the wildcard character (*) to represent all types. All permissions in the *perm_set* must be valid for all object classes in the *class_set*.

An example neverallow rule is:

```
neverallow os_user_t rubix_table_t : db_table { select insert };
```

This rules denies subjects with the *os_user_t* type to perform *select* and *insert* operations on objects of the *db_table* object class with a type of *rubix_table_t*.

DECLARING AND USING ATTRIBUTES

An *attribute* is a "tag" that may be assigned to one or more types. The tag may then be used to refer to all associated types in subsequent policy statements and rules. A type may have more than one associated attribute and an attribute may be assigned to more than one type. Once declared and associated with types, an attribute may be referred to in TE rules (by replacing the type) to define security behavior for all types associated with the attribute. For example, base SELinux policy defines an attribute named *domain*. The *domain* attribute is associated with all types that are associated with subjects. TE *allow* rules may then be added to control all domain types as a whole. To declare an attribute the *attribute* statement is used. Though not required, it is general convention that attribute names do not have any suffix (e.g., there is no "*_a*" at the end of the name).

The syntax of the attribute statement is:

```
attribute attribute_name;
```

For example, the following statement declares an attribute called *rubix_sys_table*:

```
attribute rubix_sys_table;
```

The attribute may then be assigned to types using the *type* statement as the following example shows:

```
type rubix_sys_table_t rubix_sys_table;
```

If an attribute and type have already been declared, the *typeattribute* statement may be used to assign an attribute to a type as follows:

```
typeattribute rubix_table_t rubix_table;
```

To allow a subject with the *sys_user_t* domain type to select from all tables whose types have the *rubix_sys_table* attribute, the following allow rule may be used:

```
allow sys_user_t rubix_sys_table : db_table select;
```

If there is a *sys_user* attribute associated with all system user domain types, then the following rule may be used to allow all system users (users with domain types with the *sys_user* attribute) select access to all system tables (tables with types that have the r*ubix_sys_table* attribute):

```
allow sys_user rubix_sys_table : db_table select;
```

To exclude the *os_admin_t* from the above rule, assuming the *os_admin_t* has the *sys_user* attribute, the following rule could be used:

```
allow {sys_user -os_admin_t} rubix_sys_table : db_table select;
```

RDBMS TYPE ENFORCEMENT EXAMPLE

The following diagram illustrates the TE access checks that occur as two users, Bob and Nancy, perform a *SELECT * FROM MyTab* query. Bob and Nancy have logged onto the system and set their role appropriately. Note that Bob's context has a type of *rxclient1_t* and Nancy's context has a type of *rxclient2_t*. In our example, this will cause each user to receive a different set of rows from their query.

The *MyTab* table is contained within the *MySchema* schema, which itself is contained within the *MyCat* catalog. Trusted RUBIX catalogs and schemata are analogous to operating system directories, in that their primary function is to store other objects. In order for the query to execute, the catalog and schema must be searched as follows:

Step 1: Search *'MyCat'* catalog. The catalog has a type of *'rxcat_t'*. Both users are permitted to search the catalog because of the following TE rule:

allow {rxclient1_t rxclient2_t} rxcat_t : dir search;

Step 2: Search *'MySchema'* schema. The schema has a type of *'rxschem_t*'. Both users are permitted to search the schema because of the following TE rule:

allow {rxclient1_t rxclient2_t} rxschem_t : dir search;

Next, the *MyTab* table must be opened and selected from as follows.

Step 3: Open/select from *'MyTab'* table. The table has a type of *'rxtable_t'*. Both users are permitted to perform this operation because of the following TE rule:

allow {rxclient1_t rxclient2_t} rxtable_t : db_table {use select};

Lastly, the query selects individual rows. Unlike the previous operations in this query (e.g., search catalog), where a TE *deny* security decision would cause the query to fail, a *deny* on an individual row select causes the row to be filtered from the query's result set. Note that two of the rows have a type of *rxrow1_t* and two of the rows have a type of *rxrow2_t*.

Step 4a: Bob selects individual rows. The following TE rule allows Bob to select only rows with the *rxrow1_t* type:

allow rxclient1_t rxrow1_t : db_tuple select;

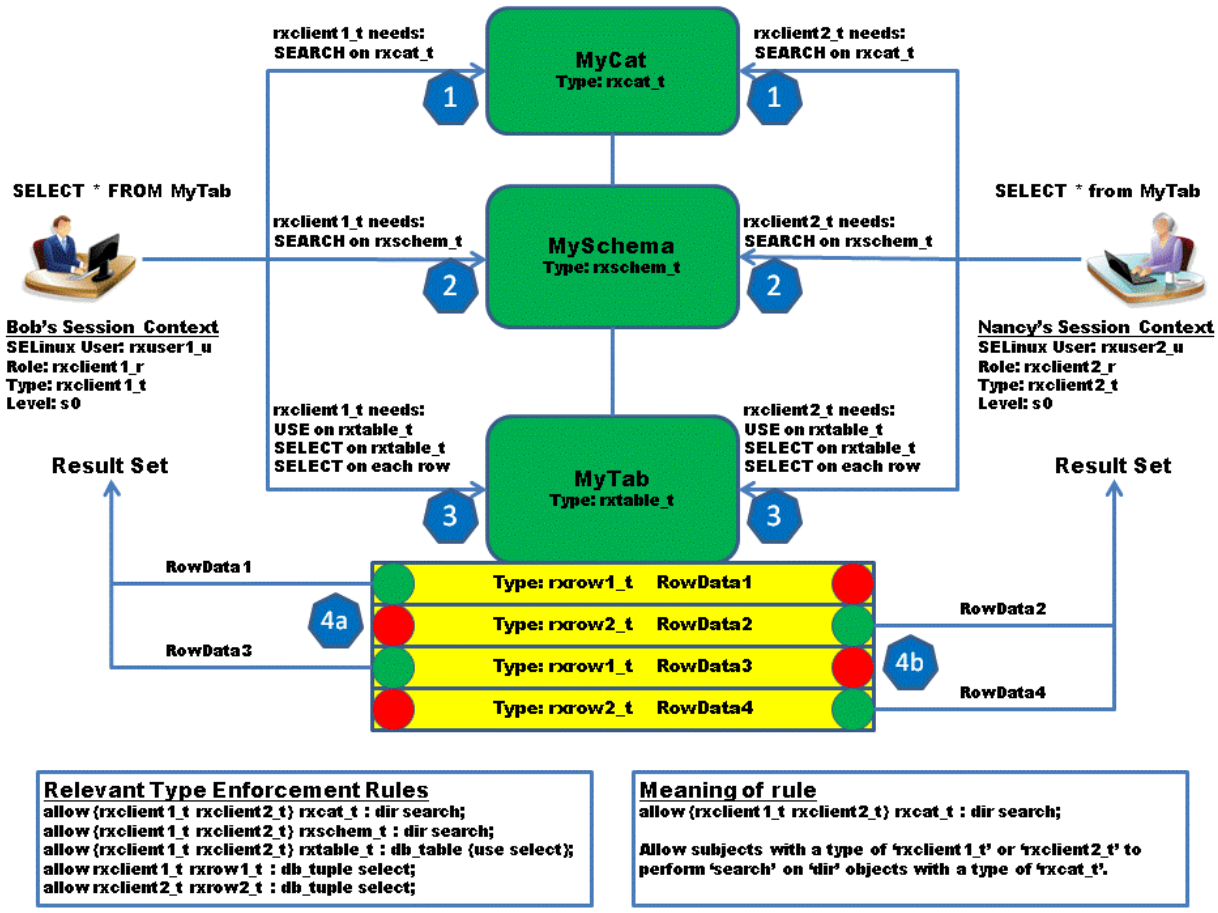
Because no TE rule exist for rows of type *rxrow2_t* being selected by a type of *rxclient1_t*, rows with that type are filtered from the result set. Bob's result set consists of *Rowdata1* and *Rowdata3*.

Step 4b: Nancy selects individual rows. The following TE rule allows Nancy to select only rows with the *rxrow2_t* type:

allow rxclient2_t rxrow2_t : db_tuple select;

Because no TE rule exist for rows of type *rxrow1_t* being selected by a type of *rxclient2_t*, rows with that type are filtered from the result set. Nancy's result set consists of *Rowdata2* and *Rowdata4*.

Figure 2: Type Enforcement During SELECT Operation



Object Labeling Rules

When an object is created SELinux calculates what context should be assigned to the new object. The level portion of the context is taken directly from the subject's session level (the low end of the subject's level range). The role is set to *object_r*, and the user is set to the creating SELinux user. The new object's type is calculated from the type of the parent object (if any) and any applicable TE rules. Note that all Trusted RUBIX RDBMS objects have a parent object except the database object and all Trusted RUBIX database objects have a fixed type of *rubix_db_t*. The TE rule used to dynamically calculate the context of a new object is called a *type_transition* rule.

If there is no applicable *type_transition* rule, the type of the parent object is chosen for the type of the new object. For instance, if a RDBMS tuple (row) is being created in a table with a type of *rubix_table_t* and there is no applicable *type_transition* rule, the type of the tuple will be *rubix_table_t*.

The syntax of the *type_transition* object labeling rule is:

```
type_transition creator_type_set parent_type_set : class_set
object_type;
```

An example of a *type_transition* object labeling rules is:

```
type_transition rubix_user_t rubix_table_t : db_tuple
rubix_user_tuple_t;
```

This rule stipulates that when a subject in the *rubix_user_t* domain creates an object with a class of *db_tuple* and the parent object has a type of *rubix_table_t* then the newly created object will have a type of *rubix_user_tuple_t*.

A file system object may also be explicitly labeled based upon its directory path using a regular expression statement. This is done using regular expression based rules. The configuration file that contains these rules for the base operating system configuration is:

```
/etc/selinux/POLICY_TYPE/contexts/files/file_contexts
```

In addition, policy modules may contain their own explicit object label rules. As an example, to label *MyFile* in the root directory the following rule may be used:

```
/MyFile    system_u:object_r:my_type_t:s0
```

To label the *MyDir* directory and all files in the *MyDir* directory the following rule may be used:

```
/MyDir(/.*)?    system_u:object_r:my_type_t:s0
```

These regular expression rules only apply to file system objects they are not applicable to Trusted RUBIX RDBMS objects.

RDBMS OBJECT LABELING EXAMPLE

The following diagram illustrates the process of calculating the label for two database rows as they are inserted by two different users. Both user's have assumed their respective roles prior to connecting to the Trusted RUBIX RDBMS. Once connected, they issue an INSERT statement, to insert distinct rows into the *MyTab* table which has a type of *rxtable_t* (the remainder of the table's context is not shown).

The first user, Bob, inserts row data *RowData1* and is shown in yellow while the second user, Nancy, inserts row data *RowData2* and is shown in red. Bob, has a RDBMS session context of *rxuser1_u:rxclient1_r:rxclient1_t:s0* while Nancy has a RDBMS session context of *rxuser2_u:rxclient2_r:rxclient2_t:s0*.

As Bob performs his insert, Trusted RUBIX requests that SELinux create a new context for the new row object. It does this by calling an operating system SELinux system call and passing in the parent

object's context (*MyTab*'s context), Bob's session context, and the object class (*db_tuple*). The SELinux system call responds with the object's new context. The new context will have the SELinux user and MLS level components copied directly from the session context. The MLS level component is taken from the low end of the level range. The row context's role is set to *object_r*, as all objects are. The type for the row's context is calculated based upon TE rules. The relevant TE rule for Bob's insert is:

type_transition rxclient1_t rxtable_t : db_tuple rxrow1_t;

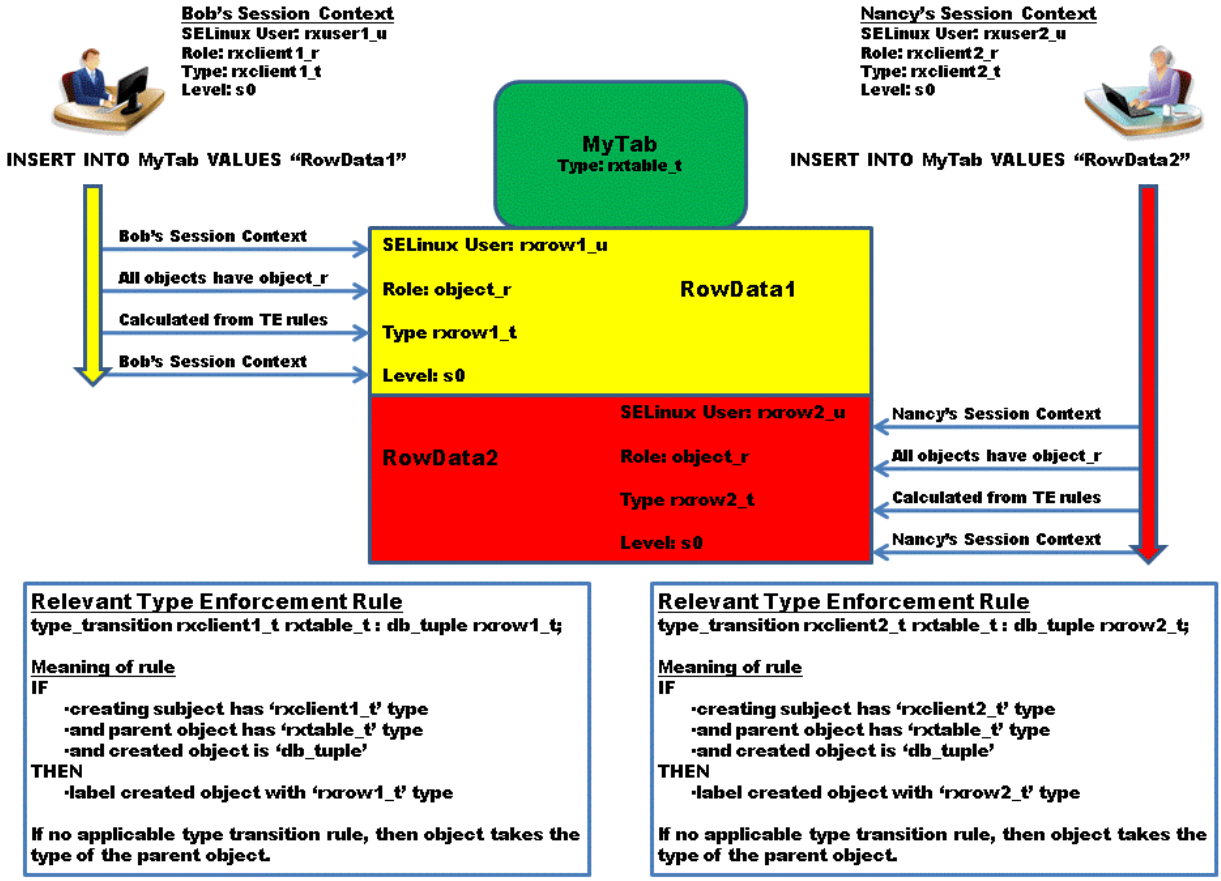
This rule says that when a type is requested for a new object of class *db_tuple*, the parent object's type is *rxtable_t*, and the creating subject's domain is *rxclient1_t* then *rxrow1_t* will be used as the new object's type.

When Nancy performs her insert, the rule above has no effect because Nancy's session type is *rxclient2_t*. The rule does not apply to her operation. However, the following rule will result in Nancy's row having a type of *rxrow2_t*:

type_transition rxclient2_t rxtable_t : db_tuple rxrow2_t;

If no *type_transition* rule applies when a RDBMS object is created it is given the same type as its parent object. This basic process is applied for RDBMS catalog, schema, table, view, and row objects. The database object always has a type of *rubix_db_t*.

Figure 3: Object Labeling During Row Insert



Domain Transitions

A domain may transition to a new domain type by executing a program configured to cause an automatic domain transition or by explicitly using the *newrole* command. The *newrole* command was previously discussed. Executing a program configured to cause a domain transition results in the executing process taking on a new domain type during the program execution. Generally, this is the method used to give trusted programs permission to access protected objects. An example of such a program is the *passwd* program. When executed the process takes on the *passwd_t* domain type which allows it to access the */etc/shadow* file. When the *passwd* program execution terminates the process is returned to its original domain type.

To cause a domain transition to be attempted when a program is executed the *type_transition* domain transition rule is used. The syntax for the *type_transition* domain transition rules is:

```
type_transition initial_domain_type exec_file_type : process
resultant_domain_type;
```

An example of the *type_transition* domain transition rule is:

```
type_transition user_t passwd_exec_t : process passwd_t;
```

This rule stipulates that when a process with the *user_t* domain type executes a program with an object type of *passwd_exec_t* then the process should **attempt** to transition to the *passwd_t* domain during the program's execution.

The *type_transition* only initiates the attempt at a domain transition. In order for the domain transition to occur the process's new domain type (*passwd_t*) must have *entrypoint* permission to the executable file's type (*passwd_exec_t*). This is given with the following *allow* rule:

```
allow passwd_t passwd_exec_t : file entrypoint;
```

Also, the original domain type (*user_t*) must have the *transition* permission to the resultant domain type (*passwd_t*). This is given with the following *allow* rule:

```
allow user_t passwd_t : process transition;
```

Lastly, the original domain type (*user_t*) must have *execute* and *gettattr* permission to the *passwd* file. This is given by the following allow rule:

```
allow user_t passwd_exec_t : file {getattr execute};
```

With regards to Trusted RUBIX, domain transitions are not an important concept for the user. Once a Trusted RUBIX user logs onto the RDBMS, no domain transitions occur as there is no concept of executing a program. The ability for a user to transition to special administrative domain types, according to the *transition* permission, is used to test if a user may operate in one of the Trusted RUBIX administrative roles.

Auditing Rules

By default SELinux audits all access checks that are denied but does not audit access checks that succeed. To override these defaults the *dontaudit* and *auditallow* rules are used.

The syntax of the *dontaudit* rule is:

```
dontaudit source_type_set target_type_set : class_set perm_set;
```

An example use of the rule is:

```
dontaudit rubix_user_t rubix_tuple_t : db_tuple select;
```

This rule stipulates that when a domain with the *rubix_user_t* is denied select on an object with class *db_tuple* and type *rubix_tuple_t* that the denial is not to be audited.

The syntax of the *auditallow* rules is:

```
auditallow source_type_set target_type_set : class_set perm_set;
```

An example use of the rule is:

```
auditallow rubix_user_t rubix_table_t : db_table use;
```

This rule stipulates that when a domain with the *rubix_user_t* is permitted to use an object with class *db_table* and type *rubix_table_t* that the operation is to be audited.

Targeted and MLS (Strict) Policies

SELinux has two policy types that may be used with Trusted RUBIX, Targeted and MLS. The Targeted policy creates TE rules for a limited number of sensitive system services and leaves all other subjects in an unconfined domain. It also implements MCS security and not MLS. Because the Targeted policy only has limited TE rule coverage, it tends to be easier to work with. That is, there are fewer abnormal or incorrect TE access denials. Since SELinux is an all encompassing security policy and the Linux operating system is very complex, the SELinux policy must also be complex. As such, the SELinux policy itself tends to be a "work in progress." Using the Targeted policy can reduce some of the frustrations of working with SELinux. The Targeted policy is used and tested far more than the MLS policy and tends to be updated frequently.

The MLS policy (or Strict policy) places every Linux operation under discrete TE rules. As such it can be difficult to work with. It is highly recommended that the user become familiar with SELinux behavior and policy modification using the Targeted policy before attempting to use the MLS policy.

The policy used by the operation system may be changed using the SELinux Management GUI tool.

It is important to note that Trusted RUBIX will enforce its RDBMS TE rules in the same manner for the Targeted and the MLS policies. The major difference is that when using the Targeted policy only MCS is usable within the RDBMS. MLS is only available when using the MLS policy. Also, **databases created under one policy may not be used when the policy is changed to the other**.

Permissive and Enforcing Modes

SELinux may be in enforcing mode or permissive mode. In *enforcing mode* the policy is enforced and operations are denied if there are no rules to permit them. In *permissive mode* no operations are denied. However, if an operation is allowed that would have been denied under enforcing mode an audit record is created. Permissive mode is useful to determine what policy rules are needed to perform an operation. The method is to place the system into permissive mode, perform the operation, and then observe any denied operations in the audit log. Rules may then be created based upon the audit messages.

When the operating system is in permissive mode, Trusted RUBIX remains in enforcing mode by default. This behavior may be changed using the configuration option *selinux.enforce.force* in the */var/lib/RUBIXdbms/etc/rxconfig* file. If this value is set to *true* (the default) the RDBMS will continue to enforce its SELinux policy while the operating system is in permissive mode. If this value is set to *false* then the RDBMS will also enter permissive mode when the operating system enters permissive mode. Regardless of the mode, Trusted RUBIX will always enforce MLS behavior. That is, permissive mode does not turn off the RDBMS MLS policy enforcement, only the Type Enforcement. This is to ensure the labeling of RDBMS objects does not become inconsistent. Please see the **Trusted Facility Manual** for more information.

Observing SELinux Denials

When SELinux denies an operations for a non-RDBMS operation a string log record is generally created in the */var/log/audit/audit.log* file. When SELinux denies a RDBMS operation a string log record is produced in the */var/lib/RUBIXdbms/logs/rxserver.log* file and */var/log/audit/audit.log* file.

The production of such log records may be controlled with the configuration options *messages.selinux.logmsg* and *messages.selinux.logrow* in the */var/lib/RUBIXdbms/etc/rxconfig* file. If *messages.selinux.logmsg* is set to *true* (the default) then RDBMS log records related to SELinux enforcement will be produced in the *rxserver.log* file; otherwise, no log records related to SELinux enforcement will be produced. If *messages.selinux.logrow* is set to *false* (the default) then rows that are filtered out of a select due to SELinux denials will not be logged in the *rxserver.log* file; otherwise such SELinux denials will result in a log record being produced (if *messages.selinux.logmsg* is also *true*). Since the SELinux denial of a row select may be a common and expected action and the potential number of such denials is large, the default behavior is not to log the denials so that the log does not become flooded.

Note that the Linux tools *ausearch* and *aureport* may be used to query the Linux audit trail.

Trusted RUBIX Roles

SELinux roles are used to authorize users to perform Trusted RUBIX administrative and SQL operations. Role configuration is performed by the *rubix-dev* SELinux policy module that is installed by the *rubix-policy-devel* package. Default role configurations of the *rubix-dev* policy are described later in this document. This policy may be customized according to local security requirements.

Note that despite authorizations assigned to a role via SELinux Type Enforcement, the user still needs to meet the additional requirements of the Discretionary Access Control (DAC), Multilevel Security (MLS), and Attribute Based Access Control (ABAC) security policies.

To use any feature of Trusted RUBIX, including connecting to a database, a user must assume a proper role. Trusted RUBIX roles were created during the installation of the *rubix-dev* policy. Trusted RUBIX roles all have a *`rubix`* prefix. Additionally, default operating system roles are conditionally given Trusted RUBIX authorizations, based upon SELinux Boolean variables. As discussed latter in this document, Boolean variables may be dynamically set on or off controlling whether the operating system roles have the authorizations.

Roles may be assumed by logging in with the desired role configured as the default or by using the *newrole* or *sudo* command. If the *newrole* command is not available you may need to install the *policycoreutils-newrole* package. The *newrole* command may also be used to change the current type or MLS/MCS level.

To transition to a new role there must be SELinux policy rules that allow transition from the source role to the target role. Additionally, the SELinux user must permit the target role. By default, the Trusted RUBIX *rubix-dev* policy allows role transactions from every operating system **login** role (e.g., *unconfined_r*, *user_r*) to every Trusted RUBIX role (e.g., *rubix_client_r*, *rubix_dbadm_r*). Therefore, the distribution of Trusted RUBIX roles may be controlled through the SELinux user configuration.

Typically, a user will login to the system and be assigned a default operating system login role. The default login roles are *unconfined_r* (Targeted policy) and *user_r* (MLS policy). The *`id –Z`* command may be used to determine the current role. The user may then use *newrole* or *sudo* to transition to a Trusted RUBIX role. The following commands demonstrate a transition from the *unconfined_r* role to the *rubix_dbadm_r* role:

```
[rxdev@RHEL6~]$ id -Z
rxdev_u:unconfined_r:unconfined_t:SystemLow-SystemHigh
[rxdev@RHEL6 ~]$ newrole -r rubix_dbadm_r
Password:
[rxdev@RHEL6 ~]$ id -Z
rxdev_u:rubix_dbadm_r:rubix_dbadm_t:SystemLow-SystemHigh
```

Using OS Roles to Operate Trusted RUBIX

The Trusted RUBIX *rubix-dev* policy gives authorizations to operating system login roles. These roles were creating by installing the operating system Targeted and MLS SELinux policies. Therefore, Trusted RUBIX may be used without assuming a Trusted RUBIX role. The authorizations are conditional upon the SELinux Boolean variables, *rubix_use_os_client_roles* and *rubix_use_os_adm_roles*. If the variables are set to *`on`* then the roles will have the Trusted RUBIX authorizations. If the variables are set to *`off`* then the roles will have no Trusted RUBIX authorizations and the Trusted RUBIX roles must be used. Examples of getting and setting a Boolean value follow (note the *root* user):

```
[root@RHEL6 ~]# getsebool rubix_use_os_adm_roles
rubix_use_os_adm_roles --> on
[root@RHEL6 ~]# setsebool -P rubix_use_os_adm_roles off
```

The following table gives an overview of the Type Enforcement authorizations given to login operating system roles by the default *rubix-dev* policy.

Operating System Role	Operations Permitted by Type Enforcement (MLS & DAC still enforced)	Boolean Variable Used to Control Authorizations
unconfined_r (Targeted policy only)	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → Import & export (*rximport*/*rxexport*)	→ rubix_use_os_client_roles
user_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → Import & export (*rximport*/*rxexport*)	→ rubix_use_os_client_roles

Operating System Role	Operations Permitted by Type Enforcement (MLS & DAC still enforced)	Boolean Variable Used to Control Authorizations
sysadm_r (Targeted policy only)	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* and *objset1* object sets → CREATE CATALOG for *default* object set → Import & export (*rximport*/*rxexport*) → Create database (*rxisql*) → Drop database (*rxdb*) → Backup database (*rxdump*) → Restore databases (*rxrestore*) → Supersede the SQL DAC policy → Operate dispatcher (*rxsvrman*) → Terminate servers (*rxsvrman*) → SPM policy management (*rxpolman*) → Session context assignment → Object context assignment → Configure auditing (*rxauditset*) → Read audit trail (*rxauditrpt*)	→ rubix_use_os_adm_roles
sysadm_r (MLS policy only)	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → CREATE CATALOG for *default* object set → Import & export (*rximport*/*rxexport*) → Create database (*rxisql*) → Drop database (*rxdb*) → Backup database (*rxdump*) → Restore databases (*rxrestore*) → Supersede the SQL DAC policy → Operate dispatcher (*rxsvrman*) → Terminate servers (*rxsvrman*)	→ rubix_use_os_adm_roles
staff_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → Import & export (*rximport*/*rxexport*) → Backup database (*rxdump*) → Operate dispatcher (*rxsvrman*) → Terminate servers (*rxsvrman*)	→ rubix_use_os_adm_roles
auditadm_r (MLS policy only)	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → Import & export (*rximport*/*rxexport*) → Configure auditing (*rxauditset*) → Read audit trail (*rxauditrpt*)	→ rubix_use_os_adm_roles
secadm_r (MLS policy only)	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → Import & export (*rximport*/*rxexport*) → SPM policy management (*rxpolman*) → Session context assignment → Object context assignment	→ rubix_use_os_adm_roles

Trusted RUBIX Default Roles

Trusted RUBIX roles are created by the *rubix-dev* policy during the installation of *rubix-policy-devel* package. The roles have a `*rubix*` prefix and may be used to operate the Trusted RUBIX RDBMS. In general, roles are created to serve as a client, database administrator, security administrator, audit administrator, and operator. The following table gives an overview of the Trusted RUBIX roles created by the default *rubix-dev* policy.

Trusted RUBIX Role	Operations Permitted by Type Enforcement (MLS & DAC still enforced)	Reachable From Role
rubix_dbadm_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* and *objset1* object sets → CREATE CATALOG for *default* object set → Import & export (*rximport*/*rxexport*) → Create database (*rxisql*) → Drop database (*rxdb*) → Backup database (*rxdump*) → Restore databases (*rxrestore*) → Supersede the SQL DAC policy → Operate dispatcher (*rxsvrman*) → Terminate servers (*rxsvrman*)	→ unconfined_r → user_r → staff_r → sysadm_r → secadm_r (MLS Policy) → auditadm_r (MLS Policy)
rubix_secadm_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → SELECT for *objset1* object set → Import & export (*rximport*/*rxexport*) → SPM policy management (*rxpolman*) → Session context assignment → Object context assignment	→ unconfined_r → user_r → staff_r → sysadm_r → secadm_r (MLS Policy) → auditadm_r (MLS Policy)
rubix_auditadm_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → SELECT for *objset1* object set → Import & export (*rximport*/*rxexport*) → Configure auditing (*rxauditset*) → Read audit trail (*rxauditrpt*)	→ unconfined_r → user_r → staff_r → sysadm_r → secadm_r (MLS Policy) → auditadm_r (MLS Policy)
rubix_op_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → SELECT for *objset1* object set → Import & export (*rximport*/*rxexport*) → Backup database (*rxdump*) → Operate dispatcher (*rxsvrman*) → Terminate servers (*rxsvrman*)	→ unconfined_r → user_r → staff_r → sysadm_r → secadm_r (MLS Policy) → auditadm_r (MLS Policy)
rubix_client_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → Import & export (*rximport*/*rxexport*)	→ unconfined_r → user_r → staff_r → sysadm_r → secadm_r (MLS Policy) → auditadm_r (MLS Policy)

Trusted RUBIX Role	Operations Permitted by Type Enforcement (MLS & DAC still enforced)	Reachable From Role
rubix_remote_client_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *default* object set → SELECT for *objset1* object set → Import & export (*rximport*/*rxexport*)	→ None. Used to label remote client sessions.
objset1_rubix_client_r	→ Client connect (*rxisql* & ODBC) → SELECT, INSERT, DELETE, UPDATE for *objset1* object set → Import & export (*rximport*/*rxexport*)	→ unconfined_r → user_r → staff_r → sysadm_r → secadm_r (MLS Policy) → auditadm_r (MLS Policy)
objset1_rubix_adm_r	→ Client connect (*rxisql* & ODBC) → All SQL DML & DDL operations for *objset1* object set → CREATE CATALOG for *objset1* object set → Import & export (*rximport*/*rxexport*)	→ unconfined_r → user_r → staff_r → sysadm_r → secadm_r (MLS Policy) → auditadm_r (MLS Policy)

Assigning Roles to Users

In order for a Linux login user (e.g., *user1*) to assume one of the Trusted RUBIX roles, the Linux login user must be mapped to a single SELinux user (e.g., *rxop_u*) and the desired Trusted RUBIX role(s) (e.g., *rubix_op_r*) must be assigned to the SELinux user. Each Linux login user must be mapped to exactly one SELinux user. A single SELinux user may be associated with many Linux login users. Any number of roles may be assigned to any number of SELinux users.

The general steps to configure Linux login users, SELinux users, and roles follows. SELinux users may be created and configured with the *semanage* command or with the SELinux Management GUI tool. The GUI tool may be started from the *System->Administration ->SELinux Management* menu. The steps are:

1. Decide the general categories of users on the system and which Trusted RUBIX functionalities and operating system functionalities they will perform. The most straight forward categories correspond to the default Trusted RUBIX roles: RDBMS client, RDBMS Administrator, RDBMS Security Administrator, RDBMS Operator, and RDBMS Audit Administrator. Alternatively, you may decide that you want all RDBMS client and administrative functionalities to reside in one category of user. This is particularly useful for a development environment. For specific steps on creating a user with all Trusted RUBIX client and administrative functionalities see the section titled **Creating a Development User** in the **Installation and Quick Start Guide**. When determining user categories, keep in mind that operating system functions may be combined with corresponding Trusted RUBIX functions. For example, you may wish to have a single Security Administrator category that may administer both the operating system and Trusted RUBIX. Or, you may wish to separate those duties into two user categories.
2. For each category of user create a corresponding SELinux user or, if applicable, use a pre-

existing SELinux user created during the operating system installation. Using the SELinux Management tool, SELinux users may be created by clicking on "SELinux Users" and clicking "Add." Additionally, the *semanage* command may be use to create and configure SELinux users. Note that by SELinux convention SELinux user names end with the "_u" suffix.

3. Assign the appropriate role(s) to each SELinux user according to the decisions made in step 1. For example, if you have an SELinux user that will need RDBMS Security Administrator functionality assign the **rubix_secadm_r** to that SELinux user. Using the SELinux Management tool, Trusted RUBIX roles may be associated with SELinux users by clicking on "SELinux Users", selecting the desired SELinux users, and clicking "Properties." The Trusted RUBIX roles may then be added to the list of roles assigned to the SELinux user. Note that you may also assign roles to an SELinux user while it is being created (i.e., step 2).
4. Decide which personnel will operate on the system and the category of user they will belong to.
5. Create a Linux login user for each person that will operate on the system. This may be accomplished using the *useradd* command or the GUI User Manager tool found under the *System->Administration->User and Groups* menu.
6. Map each Linux login user to the SELinux user that corresponds to its user category. Using the SELinux Management tool, SELinux users may be mapped to Linux login users by selecting "User Mapping" and then clicking on "Add." Additionally, the *semanage* command may be use to map users. The mapping will cause the Linux login user to assume the SELinux user (with its default role, type, and level) upon login. By default, the Targeted Policy maps all new users to the *unconfined_u* SELinux user.
7. When a role is assumed by a user, either the SELinux type must be explicitly specified or a default type must be specified in the *default_type* file in the */etc/selinux/POLICY_TYPE/contexts* directory, where POLICY_TYPE is either *targeted* or *mls*, depending on which policy you are using. It is therefore recommended that a default type mapping be added for each Trusted RUBIX role. Each Trusted RUBIX role ends with *"_r"*. The corresponding default type ends with *"_t"*. For example, the default type for the **rubix_secadm_r** role is **rubix_secadm_t**. The corresponding mapping may be set by editing the *default_type* file and adding a line as follows: "**rubix_secadm_r:rubix_secadm_t**". This should be repeated for all Trusted RUBIX roles as enumerated in the beginning of this section. Note that the installation of Trusted RUBIX *rubix-dev* policy may have performed this step for Trusted RUBIX default roles.
8. When a Linux user logs on, its user is mapped to a single SELinux user. At this time a default role is chosen to assume at login. The default role is determined by the */etc/selinux/POLICY_TYPE/contexts/default_contexts* file or a file in */etc/selinux/POLICY_TYPE/contexts/users* directory. The former case applies to all SELinux users and the latter case applies only to the SELinux user that corresponds to the file name (e.g., the *staff_u* SELinux user will have a file named *staff_u*). In order to have a default context of *user_r:user_t:s0* (if the *user_r* is available to the SELinux user) or *staff_r:staff_t:s0* (if the *staff_r* is available to the SELinux user) the following lines should be added to the file. The contexts listed first (e.g., *system_r:local_login_t:s0*) specify the login method and the list of contexts after that (e.g., user_r:user_t:s0 staff_r:staff_t:s0) specify default contexts in priority order. If none of the contexts in the context list is valid then a valid context is created arbitrarily from the group of roles and types that are valid for the SELinux user. Note that the installation of Trusted RUBIX *rubix-dev* policy may have performed this step for Trusted RUBIX default roles.

```
system_r:local_login_t:s0       user_r:user_t:s0 staff_r:staff_t:s0
system_r:remote_login_t:s0      user_r:user_t:s0 staff_r:staff_t:s0
system_r:sshd_t:s0              user_r:user_t:s0 staff_r:staff_t:s0
system_r:xdm_t:s0               user_r:user_t:s0 staff_r:staff_t:s0
```

Note that these configurations may be modified at any time.

Trusted RUBIX SELinux Policy

Overview

In order for a Trusted RUBIX installation to properly function, two distinct security behaviors must be defined through SELinux policy. The first security behavior is the proper functioning of the programs and processes that make up the Trusted RUBIX RDBMS (e.g., the *rxserver* program and process instantiations). This defines how the RDBMS interacts with the operating system and its subjects and objects. This behavior is fixed within the *rubix-base* policy installed with the RDBMS.

The second security behavior that must be defined is the proper functioning of the Trusted RUBIX RDBMS with respect to its RDBMS objects (e.g., tables and rows). This defines the particular RDBMS operations (e.g., select, insert) that any particular SELinux role may perform on a particular RDBMS object. This also defines the RDBMS administrative capabilities (e.g., back up a database, start the dispatcher) of a particular role. There is a default policy, named *rubix-dev*, installed during the Trusted RUBIX installation that defines this behavior. Therefore, the RDBMS is functional immediately after installation. This policy may be configured by the security administrator to suite the custom needs of the particular Trusted RUBIX installation. Policy script segments, called interfaces (similar to a procedure call), are included with the Trusted RUBIX policy to ease creation of custom RDBMS policies.

When Trusted RUBIX is installed two SELinux policy modules are also installed: the *rubix-base* policy module and the *rubix-dev* policy module. The *rubix-base* policy contains the basic set of SELinux policy rules that are required for the proper operation of the Trusted RUBIX RDBMS. It also contains a set of SELinux interfaces (similar to a procedure call) that supports efficient creation of custom SELinux security policies to control access to RDBMS objects. The *rubix-base* policy must always be installed and may not be modified. A copy of the policy scripts that define the interfaces provided may be found in the */var/lib/RUBIXdbms/etc/selinux/rubix-base.if* file. The file is useful for learning the exact behavior of the interfaces provided.

The Trusted RUBIX *rubix-dev* policy provides default security behavior for RDBMS objects and also is intended to be used as a starting point to create custom, on-site SELinux RDBMS security policies. The *rubix-dev* policy ensures that the Trusted RUBIX RDBMS is functional immediately after installation. In general, it provides a set of roles and Type Enforcement rules to allow those roles to perform RDBMS operations, both normal (e.g., SQL operations) and administrative (e.g., backup a database). Before a custom policy may be written the *selinux-policy-devel* package must be installed. The default *rubix-dev* policy is in the */var/lib/RUBIXdbms/etc/selinux* directory. It consists of the *rubix-dev.te*, *rubix-dev.if*, and *rubix-dev.fc* files. Instructions on building and installing a custom policy module are given later in this document.

The *rubix-base* Policy Module

The *rubix-base* policy contains the basic set of SELinux policy rules that are required for the proper operation of the Trusted RUBIX RDBMS. It also contains a set of SELinux interfaces (similar to a procedure call) that supports efficient creation of custom SELinux security policies to control access to RDBMS objects. The *rubix-base* policy must always be installed and may not be modified. A copy of the policy scripts that define the interfaces provided may be found in the */var/lib/RUBIXdbms/etc/selinux/rubix-base.if* file. The file is useful for learning the exact behavior of the interfaces provided.

RDBMS Objects and Permissions

The Trusted RUBIX RDBMS has database, catalog, schema, table, view, and row objects under SELinux control.

The TR database object is mapped to the SELinux *db_database* object class. Its permissions are *access*, *create*, and *drop*.

The TR catalog object is mapped to the SELinux *dir* object class. Its permissions are *search*, *create*, *rmdir*, *add_name*, and *remove_name*.

The TR schema object is mapped to the SELinux *dir* object class. Its permissions are *search*, *create*, *rmdir*, *add_name*, and *remove_name*.

The TR table object is mapped to the SELinux *db_table* object class. Its permissions are *use*, *setattr*, *create*, *drop*, *insert*, *select*, *update*, *delete*.

The TR view object is mapped to the SELinux *db_table* object class. Its permissions are *use*, *create*, and *drop*.

The TR row object is mapped to the SELinux *db_tutple* object class. Its permissions are *insert*, *select*, *update*, and *delete*.

The following SQL operations are controlled using Type Enforcement. The permissions required for each operation are given.

CREATE DATABASE: db_database {create}

CREATE CATALOG: db_database {access}; dir {create} on catalog

DROP CATALOG: db_database {access}; dir {rmdir} on catalog

CREATE SCHEMA: db_database {access}; dir {search add_name} on catalog; dir {create} on schema

DROP SCHEMA: db_database {access}; dir {search remove_name} on catalog; dir {search rmdir} on schema

CREATE TABLE: db_database {access}; dir {search} on catalog; dir {search add_name} on schema; db_table {create} on table

DROP TABLE: db_database {access}; dir {search} on catalog; dir {search remove_name} on schema; db_table {use drop} on table

CREATE VIEW: db_database {access}; dir {search} on catalog; dir {search add_name} on schema; db_table {create} on view; db_table {use} on any referenced table

DROP VIEW: db_database {access}; dir {search} on catalog; dir {search remove_name} on schema; db_table {drop} on view

CREATE INDEX: db_database {access}; dir {search} on catalog; dir {search add_name} on schema; db_table {use setattr} on table;

DROP INDEX: db_database {access}; dir {search} on catalog; dir {search remove_name} on schema; db_table {use setattr} on table

ALTER TABLE: db_database {access}; dir {search} on catalog; dir {search} on schema; db_table {use setattr} on table

SELECT: db_database {access}; dir {search} on catalogs; dir {search} on schemata; db_table {use select} on referenced tables; {use} on referenced views; db_tuple {select} on all rows directly selected or referenced (denied rows are filtered from the result set).

INSERT: db_database {access}; dir {search} on catalogs; dir {search} on schemata; db_table {use insert} on table/view inserted; db_table {use select} on tables/views referenced by select; db_tuple {insert} on rows inserted; db_tuple {select} on rows selected or referenced by where clause (denied rows are filtered from the result set).

UPDATE: db_database {access}; dir {search} on catalogs; dir {search} on schemas; db_table {use update} on table/view updated; db_table {use select} on tables/views referenced by select; db_tuple {update} on rows updated; db_tuple {select} on rows selected or referenced by where clause (denied rows are filtered from the result set).

DELETE: db_database {access}; dir {search} on catalogs; dir {search} on schemas; db_table {use delete} on table/view deleted from; db_table {use select} on tables/views referenced by select; db_tuple {delete} on rows deleted; db_tuple {select} on rows selected or referenced by where clause (denied rows are filtered from the result set).

RDBMS OBJECT SETS

The Trusted RUBIX policy uses a concept called *object sets* to aid in implementing coherent SELinux policy over RDBMS objects. Object sets ease in creating certain types of custom RDBMS policy. It is not required as custom policy may be created entirely from basic SELinux Type Enforcement rules.

An object set is a named set of RDBMS objects (catalogs and subordinate schemata, tables, views, and rows) that have a common security requirement. SELinux interfaces are provided that declare a named object set and to allow particular roles SQL access to the object set. For instance, if the *rubix_client_r* role is given SELECT access to the *objset_set_1* object set, then it may select from any table in the object set. It is important to note that once a RDBMS catalog has been assigned to an object set, then all **subordinate** schemata, tables, views, and rows will automatically belong to that object set.

Multiple object sets may reside in a single Trusted RUBIX database. The database object is not part of any object set. Each object set has its own unique group of SELinux object types used to control access. SELinux policy interfaces are provided to easily create object sets and to control SQL access to the object set based upon the RDBMS subjects' domain type. Interfaces exist to simply and easily permit DDL operations (e.g., object drop and create), select, insert, delete, list user objects, and list system objects based upon the domain type of the RDBMS subject and the object set being accessed. User defined object sets are named and all related SELinux constructs (e.g., roles, types) are named with a prefix equal to the object set's name.

An administrative role is created for each object set. Subjects in this role must create and drop the catalogs used by the object set.

As an example, in a cross domain environment each enclave could have a single object set to contain its RDBMS objects. Each enclave would then have a unique, named set of object types that may be used to control access. SELinux interfaces could then be used to give SQL access to a domain types for each enclave as the security requirements dictate.

Each Trusted RUBIX database has a default object set that is automatically created. In addition, it may have any number of user defined object sets explicitly created. The Security Administrator can use the provided interfaces to control access to each object set. In addition, the Security Administrator may write discrete Type Enforcement rules (e.g., *allow* rules) to further refine the security behavior.

Each object set is contained within one or more specially typed RDBMS catalogs which must be created by the object set administrator. Each object set has its own group of SELinux object types for each RDBMS object class. The SELinux types created for the default object set are *rubix_db_t*, *rubix_cat_t*, *rubix_schema_t*, *rubix_table_t*, and *rubix_row_t*. User defined object sets have object types created for each RDBMS object based upon the object set's name. For example, if an object set were created with the name *objset1* then object types would be created named *objset1_rubix_cat_t*, *objset1_rubix_schema_t*, *objset1_rubix_table_t*, and *objset1_rubix_row_t*. Additionally, an administrative role is created named *objset1_rubix_adm_r*. This role must create and drop the catalogs used by the *objset1* object set.

The sample policy source files are heavily commented to demonstrate how to use the object set concept. If discrete, custom rules are written it is recommended to review the *rubix-dev* policy source code to become familiar with using the *rubix-base* policy. The default *rubix-dev* policy is in the */var/lib/RUBIXdbms/etc/selinux* directory. It consists of the *rubix-dev.te*, *rubix-dev.if*, and *rubix-dev.fc* files.

ROLE BASED INTERFACES

The following interfaces may be used to create a Trusted RUBIX administrative role and to configure authorizations for that role. The Trusted RUBIX roles generally are able to perform administrative functions (e.g., backup a database) and/or to supersede a security policy. Please see the *Trusted RUBIX Trusted Facilities Manual* for more information on the specific authorizations of the Trusted RUBIX administrative roles.

→ **rubix_create_role**(*role_prefix*)
Create a user domain role (named *role_prefix*_r) and an associated type (named *role_prefix*_t). The created role is given no special RDBMS permissions or authorizations. It is anticipated that the role will be configured to perform RDBMS operations with one of the following interfaces.

→ **rubix_role_change_template**(*from_role_prefix,to_role_prefix*)
Allow the role specified by from_role_prefix to transition to the role specified by to_role_prefix.

→ **rubix_add_secadm_2role**(*role_prefix*)
Add Trusted RUBIX Security Administrator permissions and authorizations to the role specified by the prefix. The role is also given permissions and authorizations to be a Trusted RUBIX client. No permissions are given to perform operations on RDBMS objects.

→ **rubix_add_dbadm_2role**(*role_prefix*)
Add Trusted RUBIX Database Administrator permissions and authorizations to the role specified by the prefix. The role is also given permissions and authorizations to be a Trusted RUBIX client. No permissions are given to perform operations on RDBMS objects.

→ **rubix_add_auditadm_2role**(*role_prefix*)
Add Trusted RUBIX Audit Administrator permissions and authorizations to the role specified by the prefix. The role is also given permissions and authorizations to be a Trusted RUBIX client. No permissions are given to perform operations on RDBMS objects.

→ **rubix_add_op_2role**(*role_prefix*)
Add Trusted RUBIX Operator permissions and authorizations to the role specified by the prefix. The role is also given permissions and authorizations to be a Trusted RUBIX client. No permissions are given to perform operations on RDBMS objects.

→ **rubix_add_client_2role**(*role_prefix*)
Add Trusted RUBIX Client permissions and authorizations to the role specified by the prefix. No permissions are given to perform operations on RDBMS objects.

RDBMS OBJECT PERMISSION INTERFACES

RDBMS object permission interfaces provide a modular way to permit RDBMS SQL operations to be performed by a given domain type on a given object set. The default object set and user defined object sets have distinct sets of interfaces functions.

Default Object Set Interfaces

→ **rubix_add_admin_db_2domain**(*domain*)
Add permissions to the given domain to perform CREATE and DROP for all database objects and the ability to perform CREATE and DROP for catalog objects in the default object set.

→ **rubix_add_fullsql_dft_2domain**(*domain*)
Add permissions to the given domain to perform SELECT, INSERT, UPDATE, DELETE, LIST (i*nfo_schem* SELECT), CREATE, DROP, and ALTER for table, schema, view, index, and row objects in the default object set.

→ **rubix_add_ddl_dft_2domain**(*domain*)
Add permissions to the given domain to perform CREATE, DROP, and ALTER for table, schema, view, and index objects in the default object set.

→ **rubix_add_insert_dft_2domain**(*domain*)
Add permissions to the given domain to perform INSERT for table, view, and row objects in the default object set.

→ **rubix_add_update_dft_2domain**(*domain*)
Add permissions to the given domain to perform UPDATE and SELECT for table, view, and row objects in the default object set.

→ **rubix_add_delete_dft_2domain**(*domain*)
Add permissions to the given domain to perform DELETE and SELECT for table, view, and row objects in the default object set.

→ **rubix_add_select_dft_2domain**(*domain*)

Add permissions to the given domain to perform SELECT for table, view, and row objects in the default object set.

→ **rubix_add_list_dft_2domain**(*domain*)
Add permissions to the given domain to perform LIST (SELECT on tables in the *system_catalog.info_schem* schema) for all user objects in the default object set.

→ **rubix_add_list_sys_2domain**(*domain*)
Add permissions to the given domain to perform LIST (SELECT on tables in the *system_catalog.info_schem* schema) for all system objects.

User Defined Object Set Interfaces

→ **rubix_create_objset**(*objset*)
Create an object set with the given prefix. An administrative role (*OBJSET_rubix_admin_r*) and all RDBMS object types (e.g., *OBJSET_rubix_table_t*) are created. The administrative role may CREATE and DROP catalog objects for the new object set. The administrator must create at least one catalog prior to any other SQL operations being performed on the object set.

→ **rubix_add_fullsql_objset_2domain**(*objset,domain*)
Add permissions to the given domain to perform SELECT, INSERT, UPDATE, DELETE, LIST (i*nfo_schem* SELECT), CREATE, DROP, and ALTER for table, schema, view, index, and row objects in the given object set.

→ **rubix_add_ddl_objset_2domain**(*objset,domain*)
Add permissions to the given domain to perform CREATE, DROP, and ALTER for table, schema, view, and index objects in the given object set.

→ **rubix_add_insert_objset_2domain**(*objset,domain*)
Add permissions to the given domain to perform INSERT for table, view, and row objects in the given object set.

→ **rubix_add_update_objset_2domain**(*objset,domain*)
Add permissions to the given domain to perform UPDATE and SELECT for table, view, and row objects in the given object set.

→ **rubix_add_delete_objset_2domain**(*objset,domain*)
Add permissions to the given domain to perform DELETE and SELECT for table, view, and row objects in the given object set.

→ **rubix_add_select_objset_2domain**(*objset,domain*)
Add permissions to the given domain to perform SELECT for table, view, and row objects in the given object set.

→ **rubix_add_list_objset_2domain**(*objset,domain*)
Add permissions to the given domain to perform LIST (SELECT on tables in the *system_catalog.info_schem* schema) for all user objects in the given object set.

Utility Interfaces

→ **rubix_tcp_socket**(*type*)
Declare a TCP socket type that will be used to remotely connect to Trusted RUBIX servers.

DOMAIN ATTRIBUTES AND TYPES

The following attributes and types are assigned to Trusted RUBIX declared domains. Unless discrete, custom Type Enforcement rules are added to the *rubix-dev* policy (i.e., rules are added other than using the provided interfaces), these attributes and types may be ignored.

Domain Attributes

- → **rubix_domain_type**
  Attribute given to all domain types created through Trusted RUBIX interfaces.
- → **rubix_role_domain_type**
  Attribute given to all domain types associated with roles created through Trusted RUBIX interfaces.
- → **rubix_adm_type**
  Attribute given to all domain types associated with any administrative role created through Trusted RUBIX interfaces.
- → **rubix_dbadm_type**
  Attribute given to all domain types associated with any Database Administrator role created through Trusted RUBIX interfaces.
- → **rubix_secadm_type**
  Attribute given to all domain types associated with any Security Administrator role created through Trusted RUBIX interfaces.
- → **rubix_op_type**
  Attribute given to all domain types associated with any Operator role created through Trusted RUBIX interfaces.
- → **rubix_auditadm_type**
  Attribute given to all domain types associated with any Audit Administrator role created through Trusted RUBIX interfaces.
- → **rubix_client_type**
  Attribute given to all domain types associated with any Client role created through Trusted RUBIX interfaces.
- → **rubix_auths_domain_type**
  Attribute given to all domain types used to grant Trusted RUBIX authorizations.

Domain Types

- → **rubix_t**
  The domain type used to isolate Trusted RUBIX server and command processes. It is the creating domain type for all RDBMS system objects.
- → **rubix_secadm_auths_t**
  The domain type used to give Trusted RUBIX Security Administrator authorizations. To have the authorizations the *rubix_secadm_auths_t* domain type must belong to the current role and process domain transition must be allowed from the current domain type to the *rubix_secadm_auths_t* domain type.
- → **rubix_dbadm_auths_t**
  The domain type used to give Trusted RUBIX Database Administrator authorizations. To have the authorizations the *rubix_dbadm_auths_t* domain type must belong to the current role and process domain transition must be allowed from the current domain type to the *rubix_dbadm_auths_t* domain type.
- → **rubix_auditadm_auths_t**
  The domain type used to give Trusted RUBIX Audit Administrator authorizations. To have the authorizations the *rubix_auditadm_auths_t* domain type must belong to the current role and process domain transition must be allowed from the current domain type to the *rubix_auditadm_auths_t* domain type.
- → **rubix_op_auths_t**
  The domain type used to give Trusted RUBIX Operator authorizations. To have the authorizations the *rubix_op_auths_t* domain type must belong to the current role and process domain transition

must be allowed from the current domain type to the *rubix_op_auths_t* domain type.

→ **rubix_client_auths_t**
The domain type used to give Trusted RUBIX Client authorizations. To have the authorizations the *rubix_client_auths_t* domain type must belong to the current role and process domain transition must be allowed from the current domain type to the *rubix_client_auths_t* domain type.

RDBMS OBJECT ATTRIBUTES AND TYPES

The following attributes and types are assigned to Trusted RUBIX declared objects. Unless discrete, custom Type Enforcement rules are added to the *rubix-dev* policy (i.e., rules are added other than using the provided interfaces), these attributes and types may be ignored.

RDBMS Object Attributes

→ **rubix_dbms_type**
Attribute given to all Trusted RUBIX RDBMS object types.

→ **rubix_database_type**
Attribute given to all Trusted RUBIX RDBMS database object types.

→ **rubix_catalog_type**
Attribute given to all Trusted RUBIX RDBMS catalog object types.

→ **rubix_schema_type**
Attribute given to all Trusted RUBIX RDBMS schema object types.

→ **rubix_table_type**
Attribute given to all Trusted RUBIX RDBMS table object types.

→ **rubix_row_type**
Attribute given to all Trusted RUBIX RDBMS row object types.

→ **rubix_user_database_type**
Attribute given to all Trusted RUBIX RDBMS user database object types.

→ **rubix_user_catalog_type**
Attribute given to all Trusted RUBIX RDBMS user catalog object types.

→ **rubix_user_schema_type**
Attribute given to all Trusted RUBIX RDBMS user schema object types.

→ **rubix_user_table_type**
Attribute given to all Trusted RUBIX RDBMS user table object types.

→ **rubix_user_row_type**
Attribute given to all Trusted RUBIX RDBMS user row object types.

→ **rubix_sys_database_type**
Attribute given to all Trusted RUBIX RDBMS system database object types.

→ **rubix_sys_catalog_type**
Attribute given to all Trusted RUBIX RDBMS system catalog object types.

→ **rubix_sys_schema_type**
Attribute given to all Trusted RUBIX RDBMS system schema object types.

→ **rubix_sys_table_type**
Attribute given to all Trusted RUBIX RDBMS system table object types.

→ **rubix_sys_row_type**
Attribute given to all Trusted RUBIX RDBMS system row object types.

RDBMS Object Types For the Default Object Set

→ **rubix_db_t**
Object type given to all Trusted RUBIX database objects.

→ **rubix_cat_t**
Object type given to all Trusted RUBIX catalog objects in the default object set.
→ **rubix_schema_t**
Object type given to all Trusted RUBIX schema objects in the default object set.
→ **rubix_table_t**
Object type given to all Trusted RUBIX table objects in the default object set.
→ **rubix_row_t**
Object type given to all Trusted RUBIX row objects in the default object set.
→ **rubix_sys_db_t**
Object type given to all Trusted RUBIX system database objects.
→ **rubix_sys_cat_t**
Object type given to all Trusted RUBIX system catalog objects.
→ **rubix_sys_schema_t**
Object type given to all Trusted RUBIX system schema objects.
→ **rubix_sys_table_t**
Object type given to all Trusted RUBIX system table objects.
→ **rubix_sys_row_t**
Object type given to all Trusted RUBIX system row objects.

RDBMS Object Type Interfaces

The following interfaces may be used to mark (assign corresponding attributes) to database object types. Unless custom object types are added to the *rubix-dev* policy (i.e., types are added other than using the provided interfaces), these interfaces may be ignored.

→ **rubix_user_database_object**(*type*)
Mark the given type as a Trusted RUBIX database object.
→ **rubix_user_catalog_object**(*type*)
Mark the given type as a Trusted RUBIX user catalog object.
→ **rubix_user_schema_object**(*type*)
Mark the given type as a Trusted RUBIX user schema object.
→ **rubix_user_table_object**(*type*)
Mark the given type as a Trusted RUBIX user table object.
→ **rubix_user_row_object**(*type*)
Mark the given type as a Trusted RUBIX user row object.
→ **rubix_sys_database_object**(*type*)
Mark the given type as a Trusted RUBIX system database object.
→ **rubix_sys_catalog_object**(*type*)
Mark the given type as a Trusted RUBIX system catalog object.
→ **rubix_sys_schema_object**(*type*)
Mark the given type as a Trusted RUBIX system schema object.
→ **rubix_sys_table_object**(*type*)
Mark the given type as a Trusted RUBIX system table object.
→ **rubix_sys_row_object**(*type*)
Mark the given type as a Trusted RUBIX system row object.

RDBMS OBJECT TE RULES IN *RUBIX-BASE* POLICY

The following RDBMS object Type Enforcement rules are included in the *rubix-base* policy and may not be altered. All other RDBMS object TE rules are defined in the *rubix-dev* policy and may be modified by the

end user. These rules define how databases and system objects are labeled when they are created. All system objects are created by the *rubix_t* type and all objects created by the *rubix_t* type must be system objects. The first rule defines the type transition rule for database creation. Trusted RUBIX database objects must always have the *rubix_db_t* type.

```
type_transition rubix_client_type rubix_t : db_database rubix_db_t;

type_transition rubix_t rubix_db_t : dir rubix_sys_cat_t;

type_transition rubix_t rubix_sys_db_t : dir rubix_sys_cat_t;

type_transition rubix_t rubix_cat_t : dir rubix_sys_schema_t;

type_transition rubix_t rubix_sys_cat_t : dir rubix_sys_schema_t;

type_transition rubix_t rubix_schema_t : db_table rubix_sys_table_t;

type_transition rubix_t rubix_sys_schema_t : db_table
rubix_sys_table_t;

type_transition rubix_t rubix_table_t : db_tuple rubix_sys_row_t;

type_transition rubix_t rubix_sys_table_t : db_tuple
rubix_sys_row_t;
```

The rubix-dev Policy Module

The Trusted RUBIX *rubix-dev* policy provides default security behavior for RDBMS objects and also is intended to be used as a starting point to create custom, on-site SELinux RDBMS security policies. The *rubix-dev* policy ensures that the Trusted RUBIX RDBMS is functional immediately after installation. In general, it provides a set of roles and Type Enforcement rules to allow those roles to perform RDBMS operations, both normal (e.g., SQL operations) and administrative (e.g., backup a database). Before a custom policy may be written the *selinux-policy-devel* package must be installed. The default *rubix-dev* policy is in the */var/lib/RUBIXdbms/etc/selinux* directory. It consists of the *rubix-dev.te*, *rubix-dev.if*, and *rubix-dev.fc* files. Examining and modifying the included *rubix-dev* policy files is encouraged. Instructions on building and installing a custom policy module are given later in this document.

DEFAULT *rubix-dev* POLICY

A default *rubix-dev* policy is installed when the Trusted RUBIX RDBMS is installed. It is intended to make the RDBMS functional immediately after installation. That is, there is no requirement that custom policy be created. The default *rubix-dev* policy creates a default set of SELinux roles suitable for RDBMS operation and creates two object sets.

Default Roles of the rubix-dev Policy

Trusted RUBIX uses the SELinux Role Based Access Control (RBAC) mechanism to allow users to perform database actions, both administrative and normal. The installation of Trusted RUBIX installed a default set of roles. The default administrative roles and their general descriptions are listed below:

- → **rubix_dbadm_r:** Create, drop, backup, and restore databases; supersede the SQL DAC policy; control the dispatcher and server processes
- → **rubix_secadm_r:** Supersede the MAC policy; configure the attribute based access control SPM mechanism
- → **rubix_auditadm_r:** Configure and view the audit trail
- → **rubix_op_r:** Control the dispatcher and server processes; backup databases

The default client roles are:

- → **rubix_client_r:** Access all RDBMS objects in the default object set from a local client
- → **rubix_remote_client_r:** Access all RDBMS objects in the default object set from a remote client; SELECT only access to objects in the *objset1* object set
- → **objset1_rubix_client_r:** Access all RDBMS objects in the *objset1* object set from a local client
- → **objset1_rubix_adm_r:** Access all RDBMS objects and create/drop catalogs in the *objset1* object set from a local client

Note that administrative roles may also perform client operations and custom roles may be configured by the OS Security Administrator. For more information about the authorizations given to Trusted RUBIX administrative roles please refer to the **Trusted Facility Manual**.

Default Object Sets of the rubix-dev Policy

The *rubix-dev* policy creates two object sets. The first is called the default object set and the second is named *objset1*.

The default object set is automatically created, therefore no use of the *rubix_create_objset* interface is required. The default object set exists under the *default_catalog* catalog. Therefore, all objects contained in the *default_catalog* catalog belong to the default object set. The *rubix_dbadm_r* role may create additional catalogs which will automatically belong to the default object set. The following SELinux scripts define the SQL behavior for objects in the default object set.

- → **rubix_add_fullsql_dft_2domain(rubix_client_t):** Allow the *rubix_client_r* role full SQL abilities on objects in the default object set.
- → **rubix_add_fullsql_dft_2domain(rubix_secadm_t)**: Allow the *rubix_secadm_r* role full SQL abilities on objects in the default object set.
- → **rubix_add_fullsql_dft_2domain(rubix_auditadm_t)**: Allow the *rubix_auditadm_r* role full SQL abilities on objects in the default object set.
- → **rubix_add_fullsql_dft_2domain(rubix_op_t)**: Allow the *rubix_op_r* role full SQL abilities on objects in the default object set.
- → **rubix_add_fullsql_dft_2domain(rubix_remote_client_t)**: Allow the *rubix_remote_client_r* role full SQL abilities on objects in the default object set.

The ability to list system objects (e.g., the *Columns* table) in the Information Schema tables are not part of any object set so special interfaces are provided. The following SELinux scripts define the SQL behavior for system objects in the Information Schema.

- → **rubix_add_list_sys_2domain(rubix_client_t)**: Allow the *rubix_client_r* role to select system rows in the Information Schema.

- → **rubix_add_list_sys_2domain(rubix_secadm_t)**: Allow the *rubix_ secadm _r* role to select system rows in the Information Schema.
- → **rubix_add_list_sys_2domain(rubix_auditadm_t)**: Allow the *rubix_auditadm_r* role to select system rows in the Information Schema.
- → **rubix_add_list_sys_2domain(rubix_op_t)**: Allow the *rubix_op_r* role to select system rows in the Information Schema.
- → **rubix_add_list_sys_2domain(rubix_remote_client_t)**: Allow the *rubix_remote_client_r* role to select system rows in the Information Schema.
- → **rubix_add_list_sys_2domain(objset1_rubix_client_t)**: Allow the *objset1_rubix_client_r* role to select system rows in the Information Schema.
- → **rubix_add_list_sys_2domain(objset1_rubix_adm_t)**: Allow the *objset1_rubix_adm_r* role to select system rows in the Information Schema.

The *objset1* object set is the only named object set created by default. More object sets may be created by using the *rubix_create_objset* interface. In order to use this object set the *objset1_rubix_adm_r* role **must first create a catalog in the database**. Then, all objects created within this catalog will belong to the *objset1* object set and be controlled by its SELinux rules. Note that this role may create any number of catalogs and each will belong to the *objset1* object set. The following SELinux scripts define the SQL behavior for objects in the default object set.

rubix_add_fullsql_objset_2domain(objset1, rubix_dbadm_t): Allow the *rubix_dbadm_r* role full SQL abilities on objects in the *objset1* object set.

rubix_add_fullsql_objset_2domain(objset1, objset1_rubix_adm_t): Allow the *objset1_rubix_adm_r* role full SQL abilities on objects in the *objset1* object set.

rubix_add_select_objset_2domain(objset1, rubix_op_t): Allow the *rubix_op_r* role SQL SELECT abilities on objects in the *objset1* object set.

rubix_add_list_objset_2domain(objset1, rubix_op_t): Allow the *rubix_op_r* role Information Schema list abilities on objects in the *objset1* object set.

rubix_add_select_objset_2domain(objset1, rubix_auditadm_t): Allow the *rubix_auditadm_r* role SQL SELECT abilities on objects in the *objset1* object set.

rubix_add_list_objset_2domain(objset1, rubix_auditadm_t): Allow the *rubix_auditadm_r* role Information Schema list abilities on objects in the *objset1* object set.

rubix_add_select_objset_2domain(objset1, rubix_secadm_t): Allow the *rubix_secadm_r* role SQL SELECT abilities on objects in the *objset1* object set.

rubix_add_list_objset_2domain(objset1, rubix_secadm_t): Allow the *rubix_secadm_r* role Information Schema list abilities on objects in the *objset1* object set.

rubix_add_insert_objset_2domain(objset1, objset1_rubix_client_t): Allow the *objset1_rubix_client_r* role SQL INSERT abilities on objects in the *objset1* object set.

rubix_add_update_objset_2domain(objset1, objset1_rubix_client_t): Allow the *objset1_rubix_client_r* role SQL SELECT abilities on objects in the *objset1* object set.

rubix_add_delete_objset_2domain(objset1, objset1_rubix_client_t): Allow the *objset1_rubix_client_r*

role SQL DELETE abilities on objects in the *objset1* object set.

rubix_add_select_objset_2domain(objset1, rubix_remote_client_t): Allow the *rubix_remote_client_r* role SQL SELECT abilities on objects in the *objset1* object set.

Remote Connection Rules of the rubix_dev Policy
The ability for a role to connect to a database from a remote client must be explicitly given. The following rule gives the behavior of the policy by default.

rubix_tcp_socket(rubix_remote_client_t): Allow the *rubix_remote_client_r* role the ability to connect to a database through a TCP socket.

BUILDING AND INSTALLING CUSTOM POLICY

The OS Security Administrator may create custom policy for the Trusted RUBIX RDBMS objects. It is highly recommended that the included policy script files, in */var/lib/RUBIXdbms/etc/selinux,* be examined and modified to become comfortable with creating policy.

To build a custom policy the *selinux-policy-devel* package must first be installed. To prepare a build directory copy the contents of the */usr/share/selinux/devel* directory to the directory you wish to use. Then, copy the *rubix-dev.te*, *rubix-dev.if,* and *rubix-dev.fc* files from the */var/lib/RUBIXdbms/etc/selinux* directory into your directory. Lastly, remove the *example.te*, *example.if,* and *example.fc* files from your working directory.

The **.te* files generally contain TE rules and interface calls. The **.if* files contain your own interfaces, and the **.fc* files contain static OS file labeling rules. To see the source for the *rubix-base* policy's interface functions mentioned in this document, examine the */usr/share/selinux/develinclude/rubix-base* directory.

Trusted RUBIX policy may be constructed in five steps:

- → create administrative and client roles
- → assign system permissions and RDBMS authorizations to those roles
- → create RDBMS object sets
- → assign RDBMS object permissions to the roles
- → declare any remote socket types

SELinux policy interfaces are provided to make these steps simple. See the default *rubix-dev.te* file for a demonstration of how to perform these steps.

Once you have modified your policy files (generally only *rubix-dev.te* need be modified) you can build your policy by using the command "*make"* in your working directory. The compiled policy module may then be installed using the following command:

semodule –i rubix-dev.pp

Note that you should be the root user in the *sysadm_r* role or have the policy in permissive mode.