Trusted RUBIX™ Version 6

SELinux in Trusted RUBIX White Paper

Revision 2

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.

Printed in U.S.A.

Overview

Security Enhanced Linux (SELinux) is an open source project that integrates a general purpose Mandatory Access Control (MAC) security policy enforcement mechanism into UNIX/Linux based operating systems. It is supported by the National Security Agency (NSA) and an active open source community. It is included and enabled by default in the Red Hat Enterprise Linux (including RHEL clones, Scientific Linux and CentOS) and Fedora operating systems.

SELinux includes three distinct MAC policy mechanisms: Type Enforcement (TE), Role Based Access Control (RBAC), and Multilevel Security (MLS). SELinux uses a text-based, scripting language to configure the behavior of all policies. Each policy may be customized by security administrators by modifying, compiling, and installing the policy script associated with a particular site. The SELinux policy may cover both operating system and DBMS objects and operations.

Trusted RUBIX integrates the SELinux policies into its DBMS, providing MAC security for all DBMS objects and operations.

SELinux Context

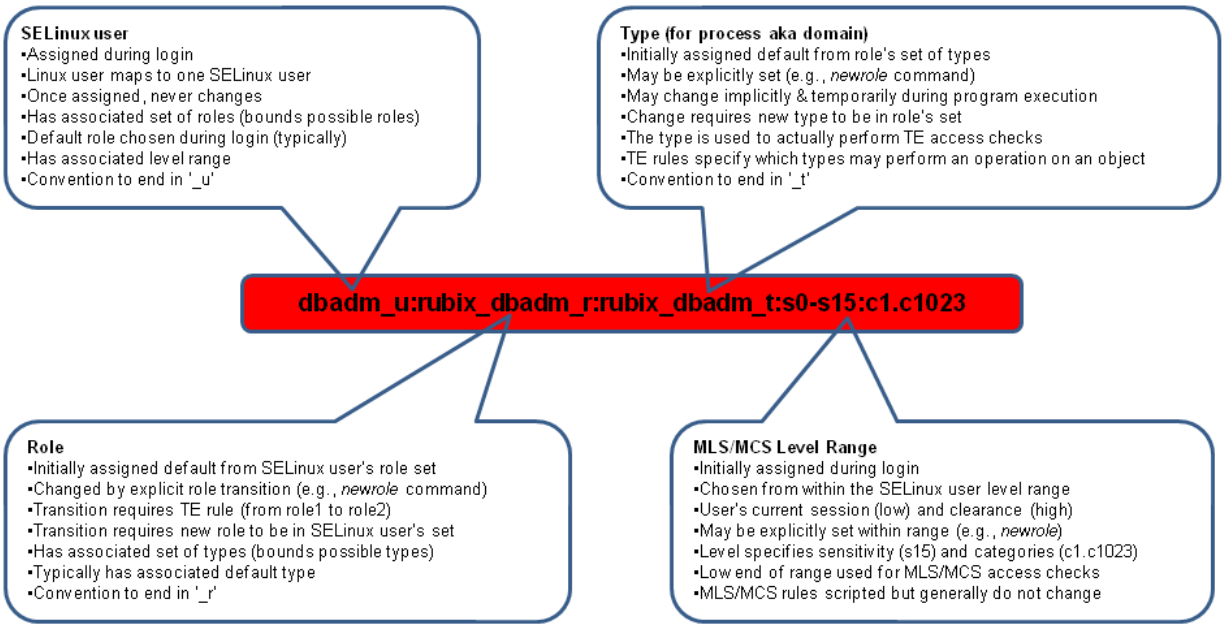
SELinux labels all of its subjects and objects with a **context**. SELinux rules may be written that determine which context an object receives at creation. Additionally, rules may be written that determine which operations a user may perform on an object given the context of the subject and the context of the object. The RBAC features of SELinux determine which contexts a given subject may acquire.

The SELinux context consists of four components:

- **SELinux User**: assigned to a Linux user upon login; bounds the user's set of available roles; never changes during a user's session; useful for auditing.
- **Role**: bounds a set of possible types; determines which role transitions may occur.
- **Type**: used to perform access check based upon the subject type, object type, object class, and operation being performed; used to write explicit access control rules.
- **MLS/MCS Level Range**: level consists of a sensitivity and group of categories; used to perform MLS access control checks.

The following diagram shows the structure of a typical SELinux context and gives further information about the characteristics of each component.

Figure 1: SELinux Context



Role Transition

The SELinux Role Based Access Control (RBAC) mechanism allows custom roles to be created and assigned to users. Each role bounds a permitted set of SELinux types that the user may assume. A user's type determines which operations it may perform. Each role bounds permitted operations over both the operating system and Trusted DBMS. Additionally, each role defines a permitted set of other roles to which the user may transition. Each role and its behavior is configured using SELinux text-based policy rules.

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.

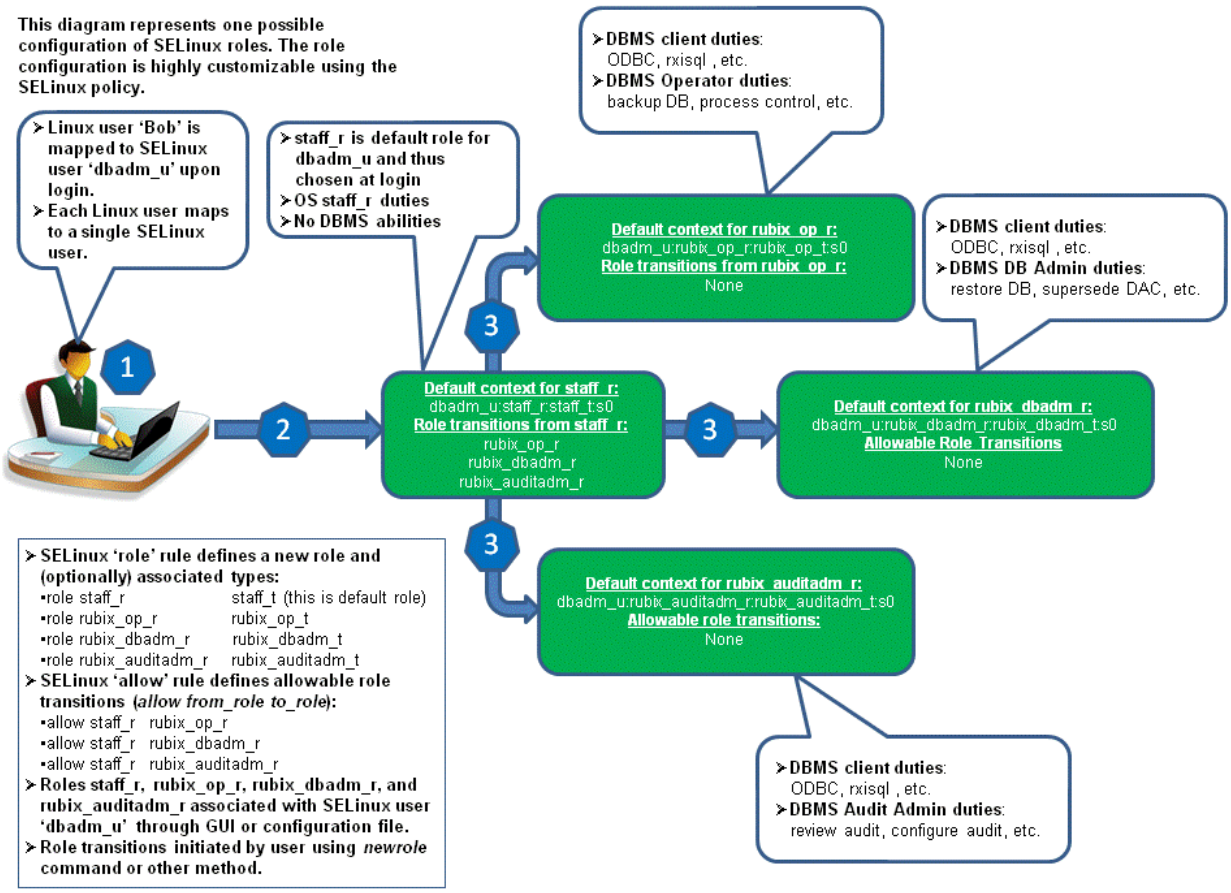
Step 1: Linux user *Bob* logs into the operating system. During login, the Linux user *Bob* is mapped to the SELinux user *dbadm_u*.

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 DBMS 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 DBMS duties, such as executing an ODBC application or performing an administrative operation (e.g., backup the database).

Figure 2: SELinux Role Transition



Type Enforcement

Type Enforcement (TE) is the SELinux mechanism that actually determines if a particular operation is permitted. In general it takes the subject's type, object's type, object's class, and operation being performed and produces a 'permit' or 'deny' security decision. Text-based policy rules determine which security decision is produced. For example, the following TE rule would allow a subject with the *rxclient_t* type to perform the SELECT operation on objects of class *db_tuple* and type *rxrow_t*:

allow rxclient_t rxrow_t : db_tuple select;

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;

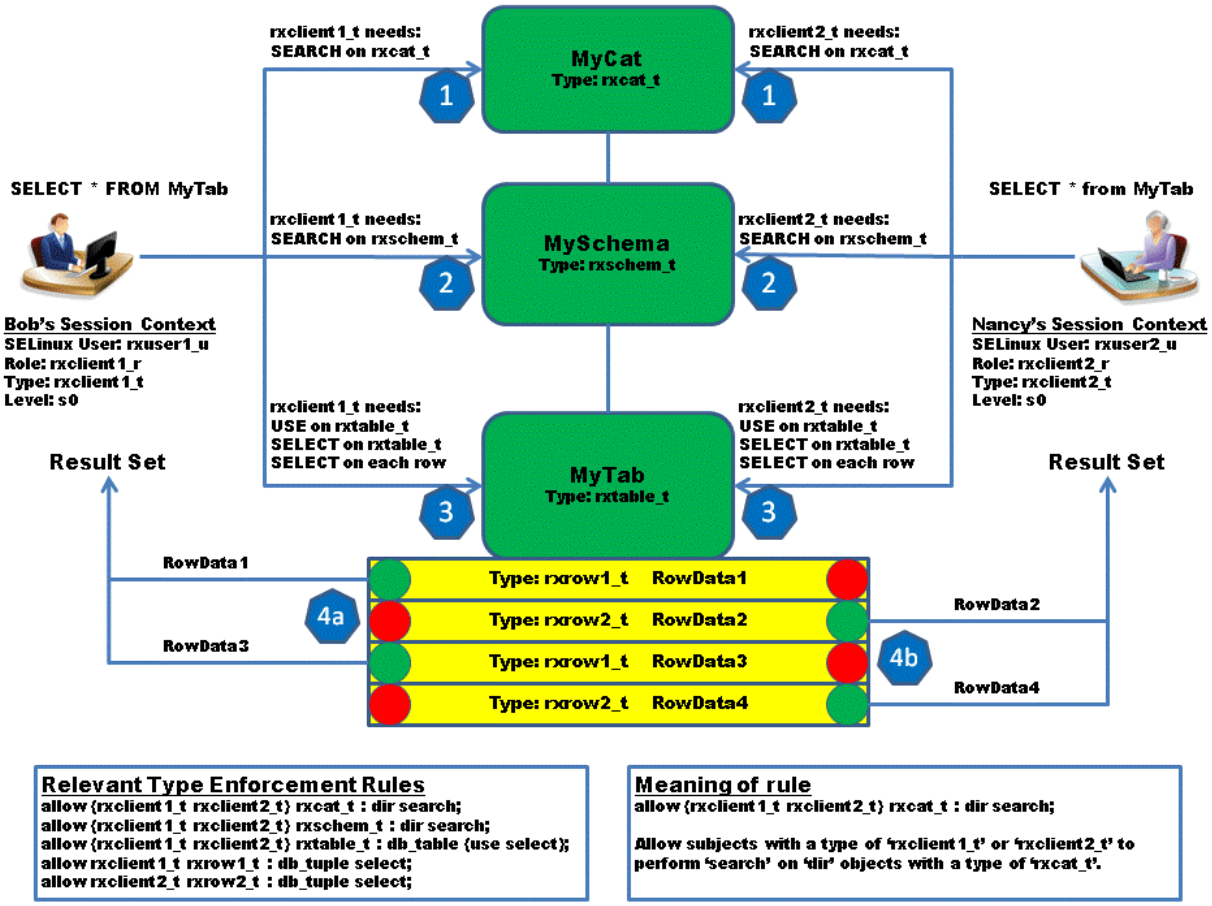
Because no TE rule exist for rows of type *rxrow2_t*, rows with that type are filtered from the result set. Bob's result set consists of *'Rowdata1'* and *'Rowdata3'*.

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*, rows with that type are filtered from the result set. Nancy's result set consists of *'Rowdata2'* and *'Rowdata4'*.

Figure 3: SELinux Type Enforcement



Object Labeling

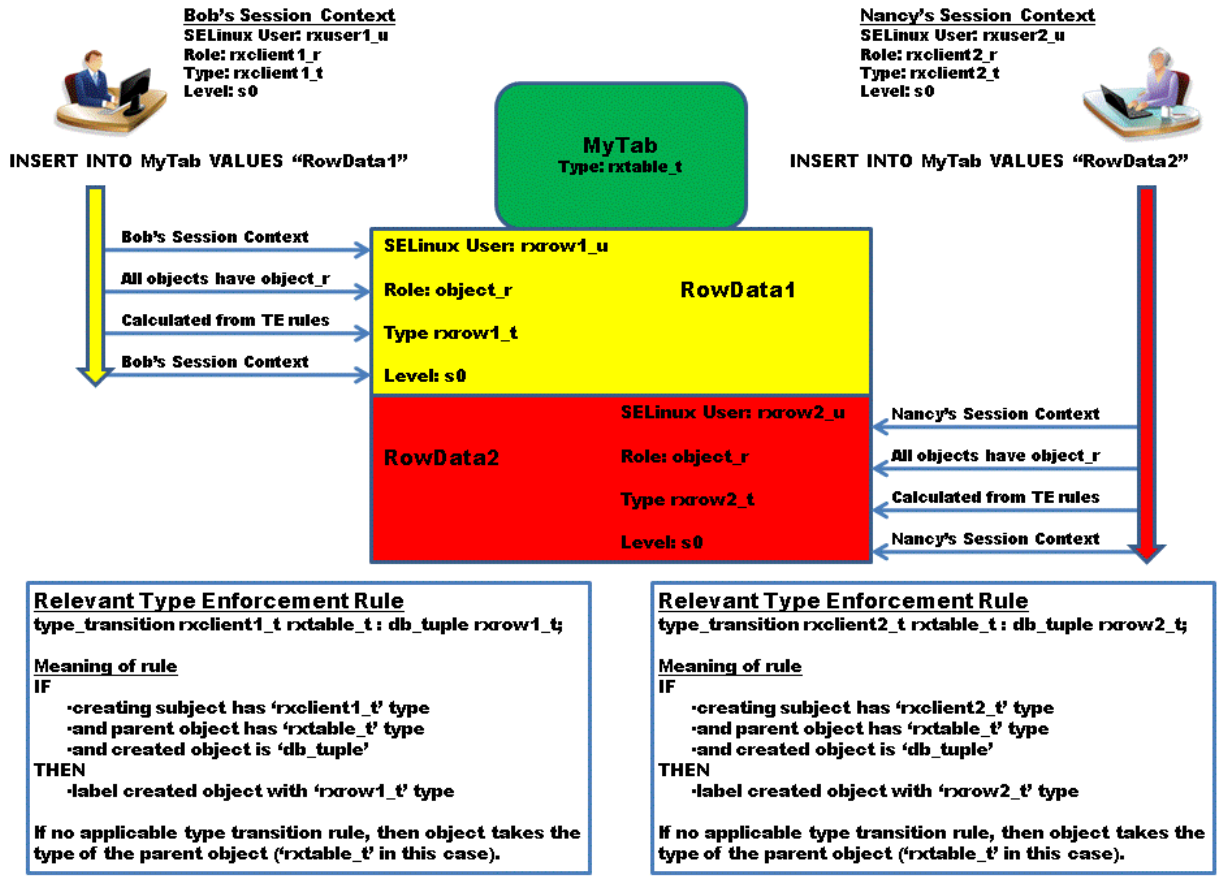
When a DBMS object is created by Trusted RUBIX (e.g., with the INSERT command), the new object's context is calculated using SELinux rules and assigned to the new object. Because this context will later be used to control access to the object using the *Type* Enforcement and MLS policies, the ability to calculate the context provides a powerful access control tool. The components of the new object's SELinux context are chosen as follows:

> **SELinux User**: Set to the SELinux User component of the creating user's session context. Useful to determine which SELinux User created the object.
>
> **Role**: Always set to the *object_r* role. The role component of an object's context is not meaningful.
>
> ***Type***: If an applicable *type_transition* rule exists, the new object's *Type* is calculated from the creating user's session context *Type* and the parent object's *Type* (e.g., the table is the parent object of a row). If no applicable rule exists then the new object takes the *Type* of the parent object. The new object's *Type* will be used to determine future *Type* Enforcement policy decisions for the object.

Level: Set to the *Level* component of the creating user's session context. The new object's *Level* will be used to determine future MLS policy decisions for the object.

The following diagram shows two examples of new row objects being labeled along with relevant SELinux rules. Bob and Nancy both insert a row into the *MyTab* table. Note that the new rows have different SELinux *Type*s (*rxrow1_t* and *rxrow2_t*) due to different session contexts and SELinux *type_transition* rules. This allows individual *Type* Enforcement rules for each row to provide distinct access control behavior.

Figure 4: SELinux Object Labeling



Process Type Transition

SELinux provides facilities (i.e., SELinux rules) to allow a Linux process to change its context *Type* during the execution of a program. Upon execution, the process's *Type* is changed to a new *Type* and upon program exit the process's *Type* is restored to the original *Type*. Typically, the new *Type* is required to perform some trusted functionality, such as checking a user's password. The result is that the trusted functionality is enabled only while the trusted program is being executed. This way the assured logic of the program is tightly bundled with the privileges needed to execute the program. If the user is not executing the program then they have no ability to perform the trusted functionality and if they are executing the program then there is better assurance that the trusted functionality will not be exploited.

Once a trusted program has been configured, SELinux rules are used to control which users may execute the trusted program, based upon the user's *Type*. Thus, the RBAC mechanism may be used to control who may perform the trusted functionalities.

An application of the SELinux process type transition is demonstrated by the following example and diagram. In the example there is a trusted program called *UpgradeInsert*. The *UpgradeInsert* program is an ODBC application that takes row data as an argument and then inserts that data as a row with an MLS level of *s1*. This occurs despite the user's process context being at an MLS level of *s0*. Thus, the program upgrades the level of the row data from *s0* to *s1*. Additionally, the upgrade is only allowed to occur if the row data meets some contextual requirements, such as the absence of "dirty words".

While there are other methods to accomplish inserting an upgraded row (see notes below), here the program transitions to the *Type* associated with the Trusted RUBIX Security Administrator role. This allows the *UpgradeInsert* program to use the Trusted RUBIX ALTER SESSION command to change the DBMS session context to have an upgraded level of *s1* prior to performing an INSERT, resulting in the new row having the upgraded level. The *UpgradeInsert* program also will perform some dirty word checks and other filtering to ensure the value being upgraded is acceptable.

In our example, the user *Bob* has the ability to execute the *UpgradeInsert* program using the trusted functionality while user *Nancy* does not. This is because *Nancy* does not have basic SELinux execute abilities for the program. Also, Nancy has no SELinux abilities to perform a process type transition. If Nancy were allowed to execute the program without a *Type* transition then the DBMS server would reject the ALTER SESSION command because the Trusted RUBIX Security Administrator role is required.

The sequential steps performed in using the *UpgradeInsert* program are:

Step 1: The user executes the *UpgradeInsert* program. The row data (e.g., *UpgradeData1*) that will be inserted is passed as the program's only argument. At this step SELinux performs a number of actions:

1) it checks that the user's *Type* has basic execute abilities for the *Type* of the program (see rule B in the diagram);
2) if an appropriate *type_transition* rule exists and the user's *Type* has rules to allow the transition, the user's *Type* will be changed to the new *Type* (see rules C, D, and E in the diagram);
3) if no *type_transition* rule exists, no *Type* change occurs prior to execution. In our example, Bob is allowed to execute the program with a type transition to *rubix_secadm_t* (as specified in rule C) while Nancy is denied execution.

Step 2: Bob's process *Type* is set to the new *Type* (*rubix_secadm_t*) and the *UpgradeInsert* program is executed. At this point, because of the *rubix_secadm_t Type*, Bob's process has the abilities of the Trusted RUBIX Security Administrator.

Step 3: The command-line argument is parsed and the row data (*UpgadeData1*) is filtered to ensure it is an acceptable value for upgrading. This functionality is entirely application dependent.

Step 4: If the row data is not acceptable for upgrading, the program exists with an error. Bob's process *Type* is automatically restored to its original value (*rxclient1_t*).

Step 5: The program connects to the Trusted RUBIX DBMS server. The DBMS session context will be extracted from Bob's process and will be *rxuser1_u:rxclient1_t:rubix_secadm_t:s0*.

Step 6: The program issues the ALTER TABLE SET LABEL SQL command to set the DBMS session label to have a level of *s1*. This step ensures that any data subsequently inserted will have a level of *s1*. Note that only a user operating in the Trusted RUBIX Security Administrator role may execute this functionality. If a

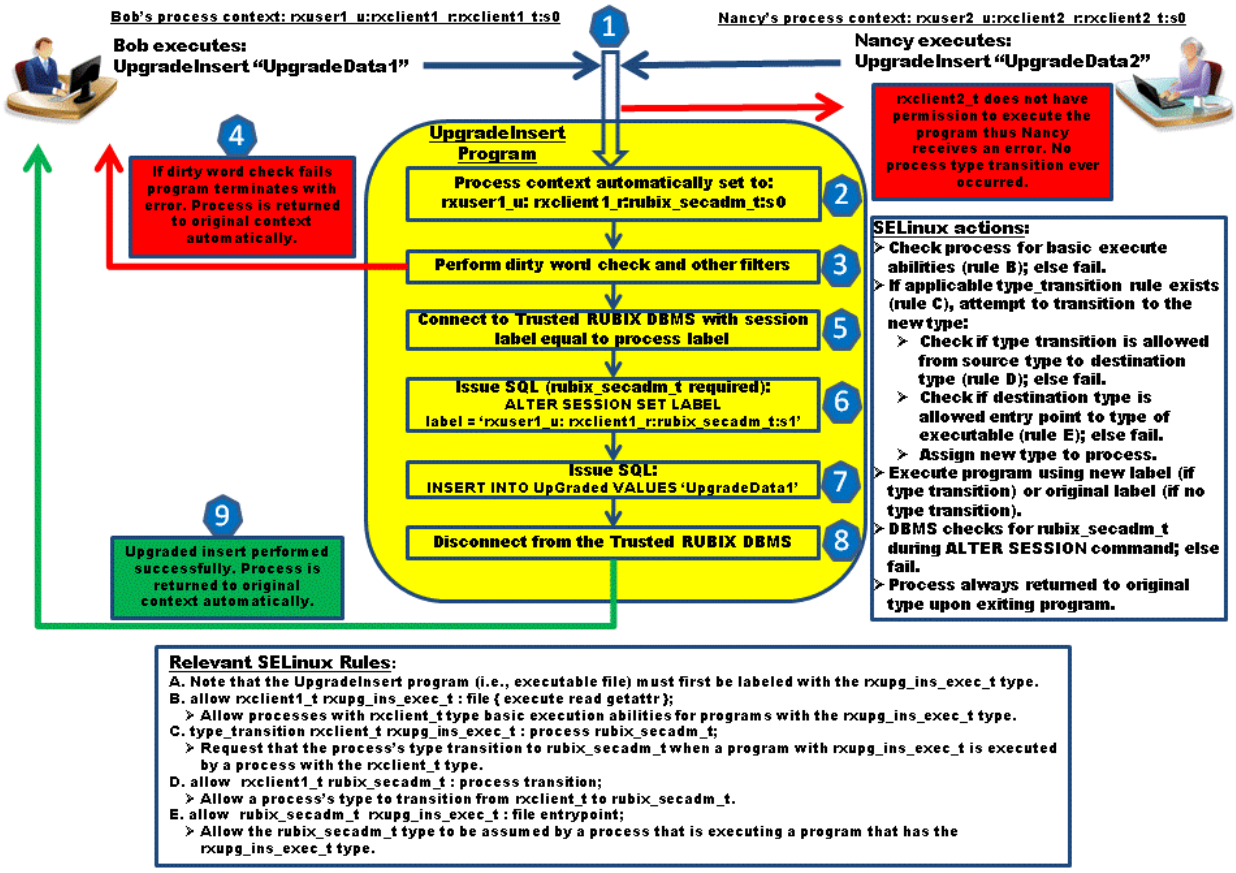
normal user was allowed to execute the *UpgradeInsert* program but there were no applicable *type_transition* rule, then the DBMS server would disallow this operation.

Step 7: The row is constructed from the *UpgradeData1* argument and inserted into the database. Because of the ALTER SESSION command issued in step 6, the new row will have a level of *s1*.

Step 8: The program disconnects from the Trusted RUBIX DBMS.

Step 9: The program exists after having successfully inserted a row at the upgraded level of *s1*. Bob's process *Type* is automatically restored to its original value (*rxclient1_t*).

Figure 5: Trusted Program (*UpgradeInsert*) Using Process Type Transition



The SELinux *range_transition* rule will allow a process's Level to transition during a program's execution. This is as opposed to the *Type* transition (*type_transition* rule) used in the example. Had the *range_transition* rule been used, the process would transition directly to the *s1* level and the row would be inserted. There would be no need for the ALTER SESSION command. The current method was chosen for the example to demonstrate the use of a Trusted RUBIX administrative role. So, the example shown could be modified to make use of the Trusted RUBIX DBA role to allow the program to supersede DBMS DAC.

A similar example could also have been used where the new *Type* of the process transition allows DBMS operations to be performed that would not be permitted by the original *Type*. For example, the process could transition from *rxclient1_t* to *rxclient_trusted_t* and then SELinux *Type* Enforcement rules would

allow only *rxclient_trusted_t* to perform the trusted DBMS operations (e.g., SELECT, UPDATE).

Trusted RUBIX Object Set Model

The Trusted RUBIX policy uses a concept called ***object sets*** to aid in implementing coherent SELinux policy over DBMS objects. Object sets ease in creating certain types of custom DBMS 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 DBMS 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 DBMS 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 DBMS 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 DBMS 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.

As an example, in a cross domain environment each enclave could have a single object set to contain its DBMS 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 DBMS catalogs which must be created by the object set administrator. Each object set has its own group of SELinux object types for each DBMS 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 DBMS 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*.

Example: Cross Domain using Type Enforcement

This example demonstrates the basic capabilities of the SELinux Type Enforcement (TE) policy in a Cross Domain environment. Two fundamental Type Enforcement capabilities are demonstrated:

The ability to calculate a database object's context (i.e., its *Type*) based upon the domain that creates the object (e.g., *US Domain*) and the container of the created object (e.g., a table is a container of a row).

The ability to permit or deny SQL operations upon an object (e.g., row) based upon the operation (e.g., SELECT, UPDATE), the domain that requests the operation, and the object's context (e.g., the row's *Type*).

These TE capabilities are configured using a simple scripting language and may be customized for a particular environment.

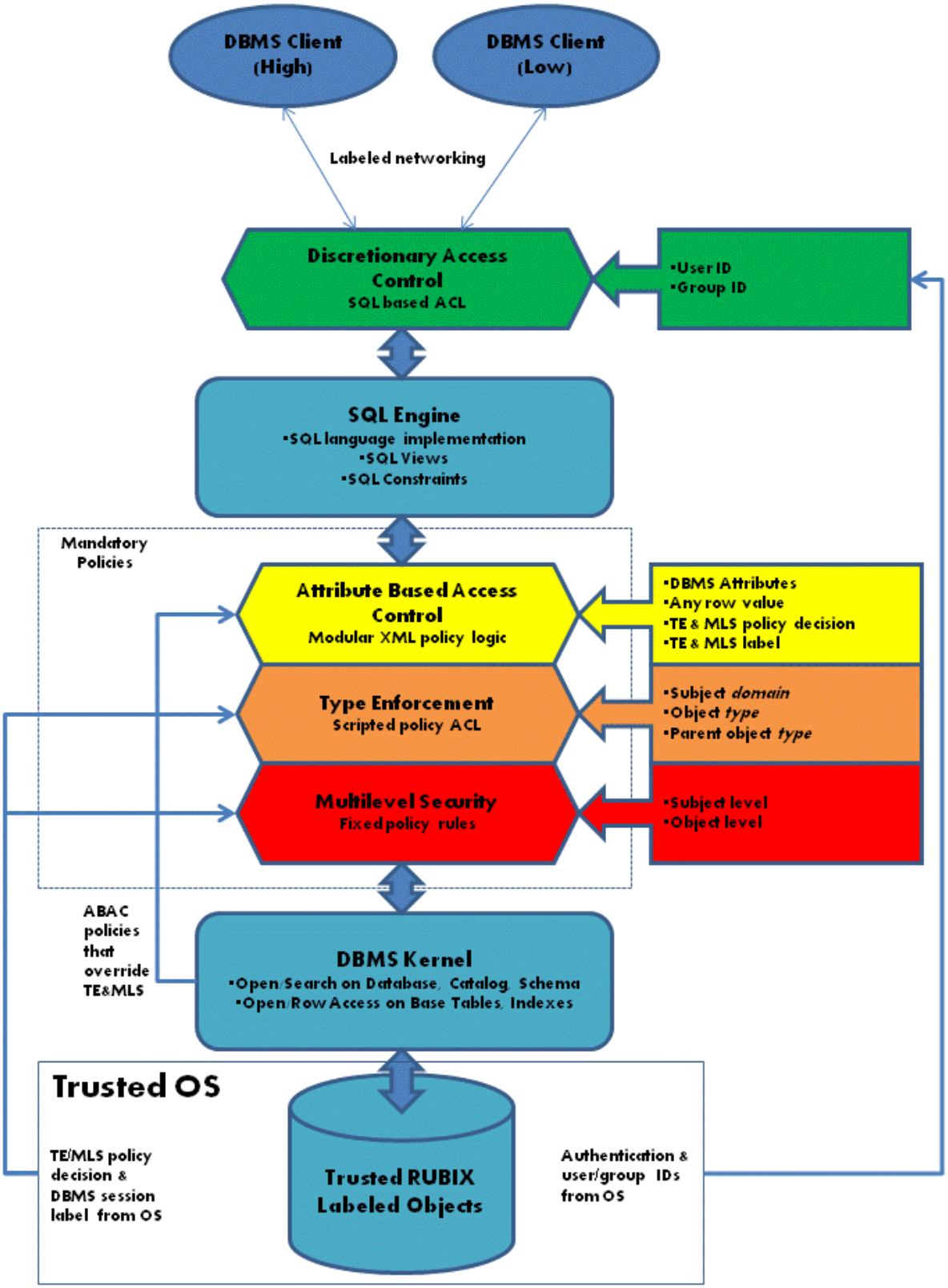
Note that these TE capabilities are not possible using the more static Multilevel Security (MLS) policy model. The MLS model predetermines an object's creation label to be equal to the label of the creating subject. Therefore, with MLS, a subject may only create objects with a single label while Type Enforcement allows objects with any number of unique contexts to be created. Because the creation contexts are used for access control, the plurality of object contexts allow very flexible access control decisions. Additionally, MLS predetermines the behavior between objects and subjects based upon their labels (i.e., write objects with equal labels and read objects with less than or equal labels). With TE an Access Control List (ACL) approach is taken where the security behavior of each SQL operations (e.g., SELECT, UPDATE) is defined using the scripting language.

The security requirements of the example and related SELinux script rules are:

- A single cross domain table named *FlightArrivals* containing one row for each arrival of an aircraft at a US airbase. Each row will have the aircraft type and the flight's origin.
- The table has a *Type* of *usarrivals_t*.
- Three security domains: United States (US), United Kingdom (UK), and France (FR). Flights only arrive from the US and UK. France will only be able to observe flight arrivals originating from the UK.
- Trusted RUBIX DBMS session will be labeled *usdom_t, ukdom_t,* or *frdom_t* if the session originates from the US Domain, UK Domain, or FR Domain, respectively.
- The US and UK Domains may insert rows into the table representing flights originating from their respective countries. As rows are inserted they are automatically labeled with a *Type* identifying the domain (*usflt_t* and *ukflt_t*). The France Domain is not permitted to insert rows. The following SELinux type_transition rules cause the rows to automatically be labeled with the appropriate Type.
    - *type_transition usdom_t usarrivals_t : db_tuple usflt_t*
    - *type_transition ukdom_t usarrivals_t : db_tuple ukflt_t*
- The US Domain may select all rows. The UK and FR Domains may only select rows representing flights from the UK (i.e., rows with the *ukflt_t Type*).
    - *allow usdom_t usflt_t : db_tuple select*
    - *allow usdom_t ukflt_t : db_tuple select*
    - *allow ukdom_t ukflt_t : db_tuple select*
    - *allow frdom_t ukflt_t : db_tuple select*
- The US and UK domain may only update rows which were inserted from their respective domain (*usflt_t* and *ukflt_t Types* respectively). The France domain is not permitted to update rows.
    - *allow usdom_t usflt_t : db_tuple update*
    - *allow ukdom_t ukflt_t : db_tuple update*
- The US domain may delete any row as the corresponding flights terminate. The UK and France domains are not permitted to delete rows.
    - *allow usdom_t usflt_t : db_tuple delete*
    - *allow usdom_t ukflt_t : db_tuple delete*

The following diagram shows the architecture and SQL operation behavior.

Figure 6: Cross Domain Using Type Enforcement



Appendix A: Frequently Asked Questions (FAQ)

1) What security policy (for SELinux only) provides user sessions that are assigned domains and TR objects that are assigned Types (labels)?

In TR, the Type Enforcement (TE) model provides highly granular access control. A scripting language is used to define which type is assigned to an object and which domain is assigned to a user session. This provides more flexible policy expressions than those of MLS access controls.

2) How does TE work in TR?

Each user and data item is assigned a label (type) and rules are written to explicitly allow a user of a specific type to perform a specific operation (e.g., SELECT) on data of a specific type. Rules are also written to define which type is used to label the data when it is created and which type a user assumes when logged in to TR. TE base policy includes the ability to control access to TOS and TR objects. TE rules allow users to transition from one type to another.

3) How do users apply MAC to TR objects?

TR applies MAC (MLS + TE) to all TR objects automatically, including the data dictionary. TR MAC covers all objects and operations, including SQL operations and administrative operations, e.g., database backups, audit, etc.

4) How does the concept of Role Based Access Control (RBAC) relate to TE in a SELinux environment?

In the traditional context of non TOS, RBAC specifies the access that users in certain roles may receive. In SELinux, role-based access is specified in terms of TE. So, the goal is to allow management of privileges based on roles that the authorized user may assume. Also, domains of influence that a role may enter by specifying the TE domains with which a role may be combined into a "valid context" are "restricted".

5) How do the RBAC features of SELinux determine to which context a given subject may acquire?

The SELinux context consists of four components:

- **SELinux User**: assigned to a Linux user upon login; bounds the user's set of available roles; never changes during a user's session; useful for auditing.
- **Role**: bounds a set of possible types; determines which role transition may occur.
- **Type**: used to perform access checks based upon the subject type, object type, object class, and operation being performed; used to write explicit access control rules.
- **MLS/MCS Level Range**: level consists of a sensitivity and group of categories; used to perform MLS access control checks.

6) How does the RBAC mentioned on the TR web site (www.rubix.com) relate to SELinux (TOS) RBAC?

The RBAC mentioned on the web site is for distribution of administrative privileges only and is not configurable. It does not provide an access control over data access. SELinux does have an RBAC concept and it does control access to data, but it is tightly integrated with TE and SELinux as a whole.