Trusted RUBIX™ Version 6

Attribute Based Access Control and Security Policy Manager 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

Printed in U.S.A.

Overview

The Trusted RUBIX Security Policy Manager (SPM) is a mechanism to enforce flexible and dynamic Attribute Based Access Control (ABAC) security policies during the operation of the Trusted RUBIX DBMS.

For detailed information on the Trusted RUBIX SPM, please see the Trusted RUBIX Security Policy Manager Reference Guide and the Trusted RUBIX Security Policy Manager Tutorial.

The SPM mechanism allows access control to the row level, based upon the ID of an application (app) user, plus other attributes. The app user mechanism extends the I&A of Trusted RUBIX to the app user, e.g., web user. The I&A of the app user and the access control of the SPM work together to secure transaction based web applications, e.g., Internet banking.

Security policies are created using the XML based Trusted RUBIX Security Markup Language (RXSML). The RXSML language allows policy creation and execution using a host of context attributes and functions to manipulate them. The RXSML language also allows actions, called obligations, to be executed based upon the outcome of the security policy execution. Policies may be configured to release information across any domain defined by the underlying operating system's Mandatory Access Control policy.

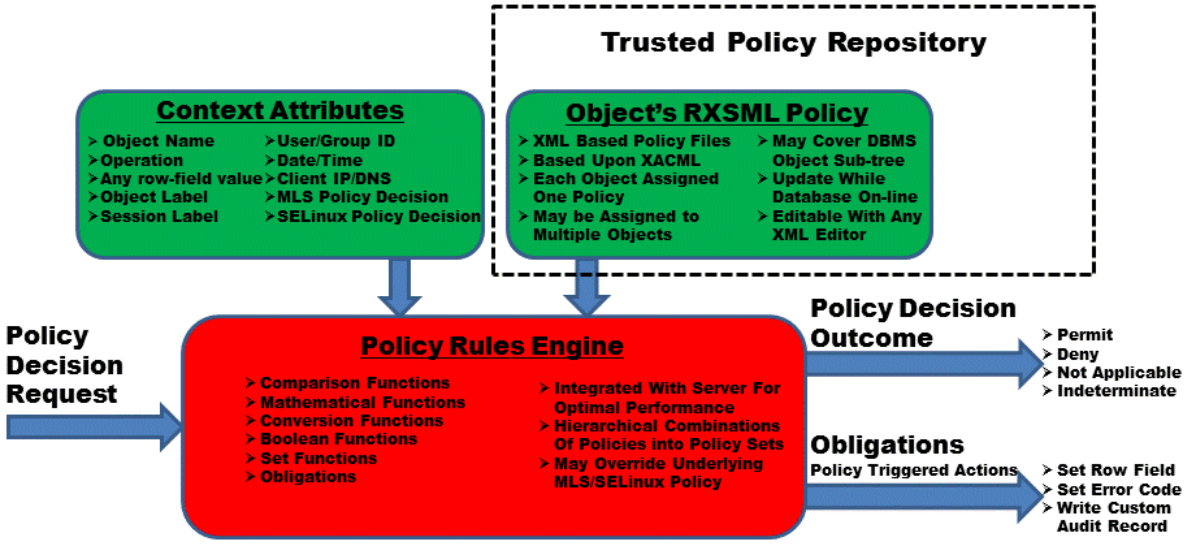
The RXSML language is based upon the policy language of the OASIS XACML 2.0 standard.

Access control logic code is organized into rules, policies, and sets of policies and algorithms may be specified to define how they interact with each other. Policies and policy sets may be referenced by name allowing for the elegant, modular design of complex policy logic and the reuse of policy logic without code duplication. Policies are assigned to DBMS objects and may be specified to protect a single object or an entire subtree of objects. Policies may also be configured to automatically protect newly created objects.

Security Policy Manager Architecture

The following diagram shows the architecture of the Trusted RUBIX (TR) SPM mechanism. When a TR operation is performed (e.g., a SELECT operation), a Policy Decision Request is made for each object. The decision is reached by executing all applicable Trusted RUBIX Security Markup Language (RXSML) policy code associated with the TR objects being acted upon. The RXSML policy will use some number of context attributes (e.g., user name, session label) to reach its decision. The Policy Rules Engine will execute the policy logic and operations over the set of context attribute values and an outcome will be reached (e.g., Permit, Deny). In addition to the policy outcome, the Policy Rules Engine may perform security critical actions called obligations (e.g., write a custom audit record), as defined by the policy.

Figure 1: Security Policy Manager Architecture



Associating Policy with DBMS Objects

Trusted RUBIX ABAC policies are stored as text-based XML files. They may be created and edited with any XML editor, on any platform by any user.

Once the XML policy files have been finished, they would generally be submitted to a Security Administrator for review. The Security Administrator would review the policy files for correctness and then add them to the Trusted RUBIX Policy Repository using the *rxpolman* Trusted Administrative command. The Policy Repository is a trusted location to store and protect approved policies. The contents of the Policy Repository may be modified only by the Security Administrator using the *rxpolman* command. A protected Policy that is in the Policy Repository but has not been applied to a Trusted RUBIX object will not have any effect upon the database.

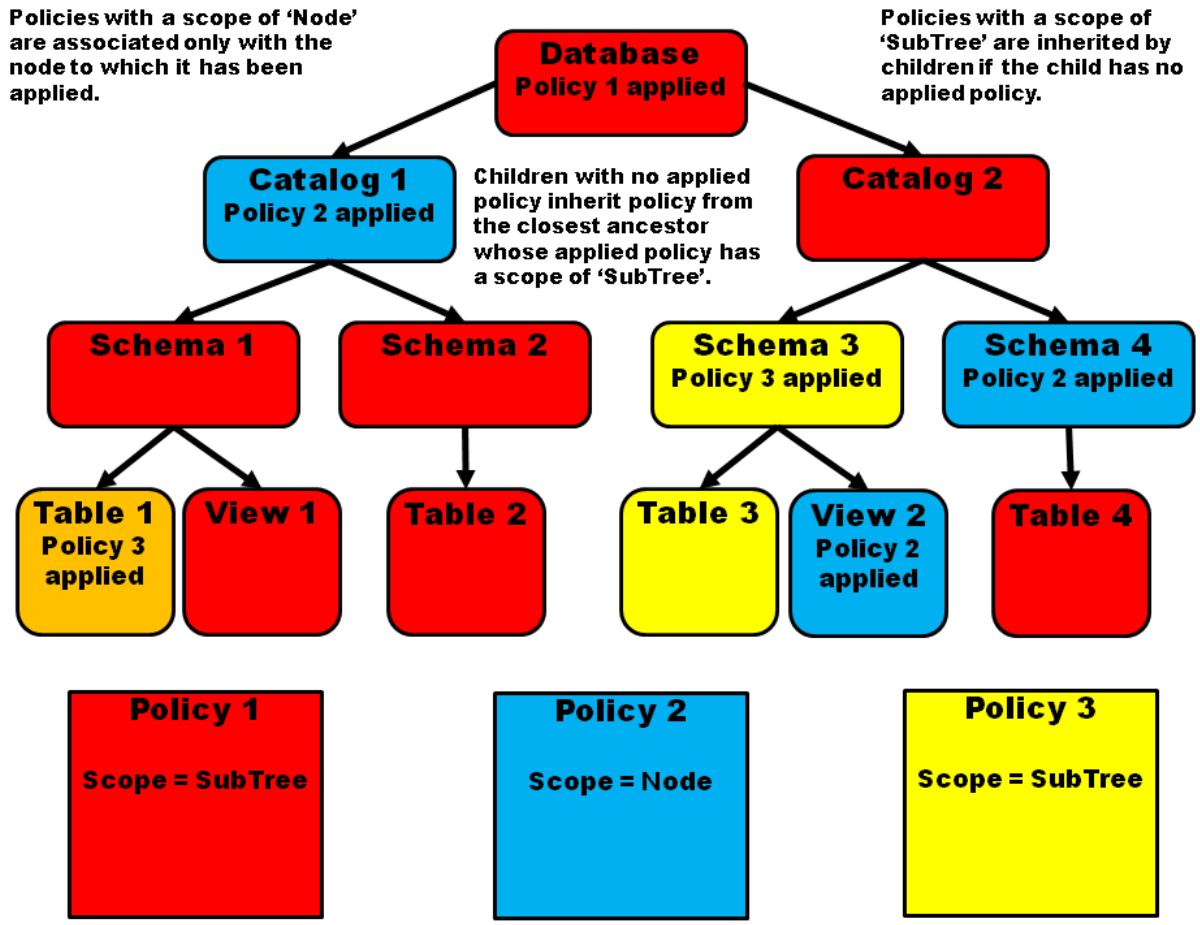
Once an XML policy file has been added to the Policy Repository, it may then be applied to Trusted RUBIX objects. Policies are applied to named Trusted RUBIX objects by the Security Administrator using a Trusted Administrative command. Once applied, policies will immediately begin controlling access to associated objects. Policy may be applied and removed in real time allowing dynamic policy behavior.

Each policy has an attribute indicating the scope of the policy. A policy scope may either be *Node* or *SubTree*. If a policy has a scope of *Node*, then it will only control the object to which it has been applied.

If the policy has a scope of *SubTree*, then it will control all objects in its subtree that do not have directly applied policy. An object is controlled by the *SubTree* scoped policy that has been applied to its closet ancestor.

The following diagram illustrates applying policy with scopes of both *Node* and *SubTree*. The color of each object node corresponds to the policy that controls it. For example, the *Schema 3* object is yellow which indicates it is controlled by *Policy 3* which is also yellow.

Figure 2: Security Policy - DBMS Object Association



Policies and Policy Sets

The main constructs of the Trusted RUBIX Security Markup Language (RXSML) are the Policy and Policy Set. Each is a named, top-level XML element and every RXSML file must contain either a Policy or Policy Set as the top-level element. Furthermore, Policies and Policy Sets are the construct which is applied to Trusted RUBIX (TR) objects.

A Policy is the RXSML construct that contains policy rules and attribute level logic and their associated functions. Policies may contain more than one rule, in which case an algorithm is chosen to define how they interrelate. For example, the *Ordered Permit Override* will execute each rule in order and the first Permit evaluation will cause the Policy to evaluate to Permit.

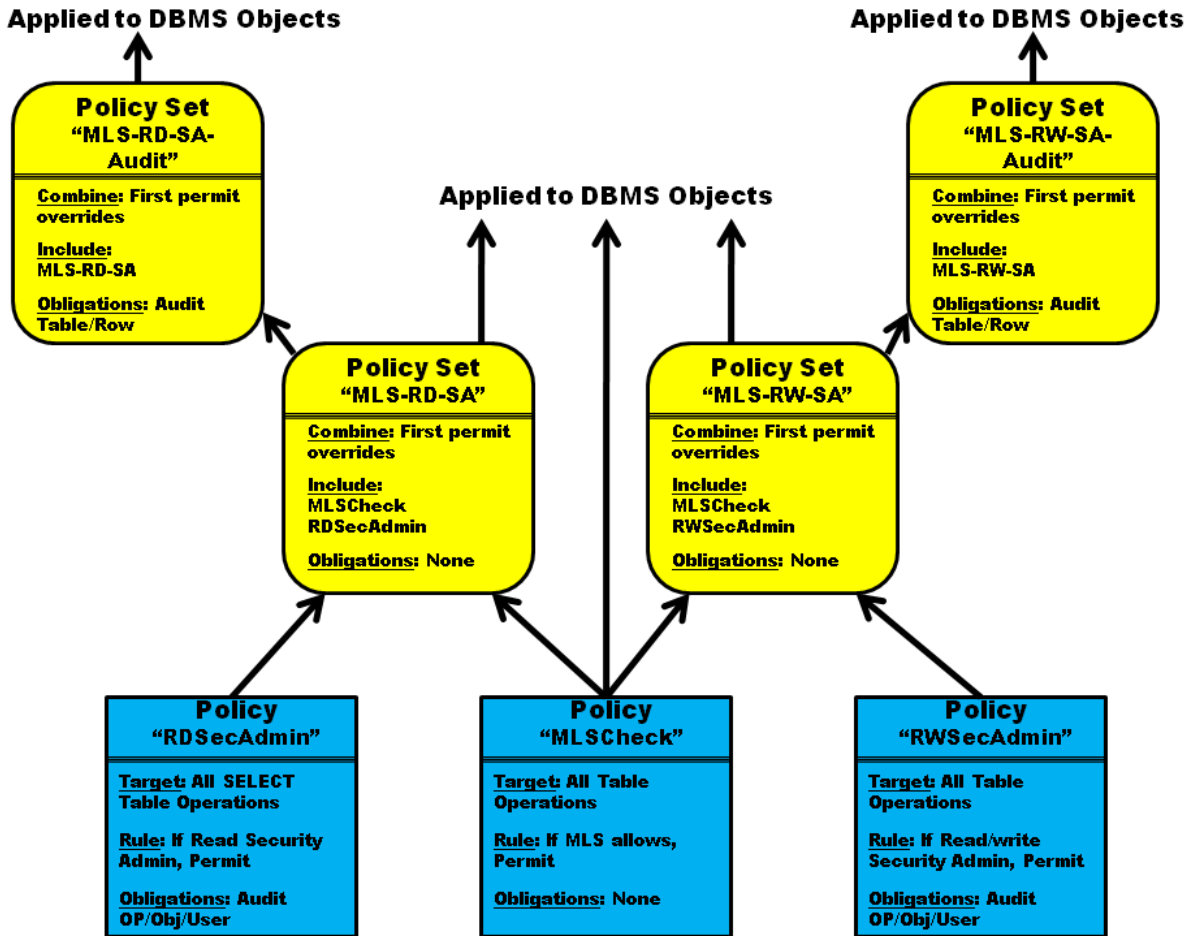
A Policy Set contains other Policies and an algorithm to define how they interrelate. Policies may be included into a Policy Set by named reference or explicitly. When included by reference the Policy may reside in a separate RXSML file. Policy may be included by reference into more than one Policy Set, allowing for modular policy design and efficient code reuse.

Policies and Policy sets have a specified target. The target specifies which subjects, objects, and actions are controlled by the policy.

Security relevant actions (e.g., setting a row field), called **obligations**, may be associated with both Policy and Policy Sets. The actions may be configure to execute when the policy evaluates to Permit or Deny.

The following diagram shows multiple Policies (light blue) and Policy Sets (yellow) organized to allow Multilevel Security override to a read-only and read-write Security Administrator. The top-level Policy or Policy Set that contains all of the needed security logic are applied to TR objects.

Figure 3: Policies and Policy Sets



Attribute Values

Context attribute values are the data values that serve as input to the policy logic. Attributes are grouped into four categories: subject attributes, resource attributes, action attributes, and environment attributes. The value of each attribute is extracted at the time of policy evaluation. Attributes are typed values and

may be manipulated using multiple functions.

The table below lists the context attributes supported by the Trusted RUBIX Security Policy Manager.

Table 1: Supported Context Attributes

Subject Attributes	Resource Attributes	Action and Environment Attributes
• subject-id • subject-name • group-id • group-name • session-start-time • session-start-date • session-start-dateTime • ip-address • dns-name • session-label	• resource-label • resource-name • row-label • table-label • view-label • schema-label • catalog-label • database-label • column-name • table-name • view-name • schema-name • catalog-name • database-name • any row-field value	• action-id • action-type • current-time • current-date • current-dateTime

Using Database Rows as Attributes

The value of any database row, including the current row being operated upon, may be used as a context attribute. This provides great power and flexibility in creating adaptive Trusted RUBIX ABAC policy.

For sample usage please see **Example: Individual Tables with IP Address White Lists** and **Example: Row Access Restricted to Row Creator** latter in this document.

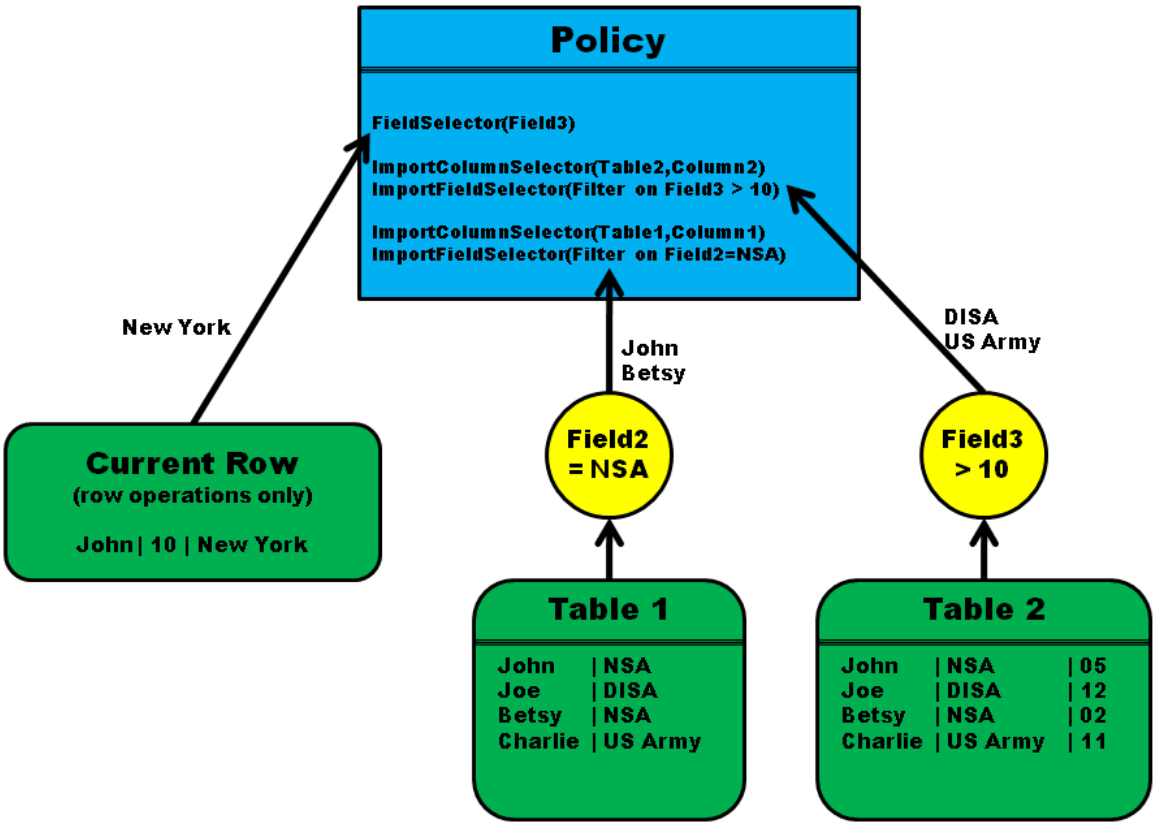
A single field of the current row being operated upon may be extracted into policy using the *FieldSelector* operator. During execution this will be replaced with the value of the specified row field. Multiple *FieldSelector* operators may be used within a single policy.

A set of column values from any database table may be imported into Trusted RUBIX ABAC policy using the *ImportColumnSelector* operator. During execution this will be replaced by a set of values from the specified table and column. The set of values may optionally be filtered using the *ImportFieldSelector* operator. This operator will allow any field value from the table row to be used as input to a boolean expression. The result of the boolean expression will determine if the row is included in the imported set of values.

When rows are imported from database tables and used to make security decisions within a policy, special care must be taken to protect the modification of the rows within the tables. Typically, RXSML policy would be deployed that restricts modification to the Security Administrator.

The following diagram shows a policy which uses *Field3* of the current row, *Column1* from *Table1,* and *Column2* from *Table2* as attributes. Note that the filter applied to a table column may use any field of the row.

Figure 4: Rows Imported as Attributes



Policy Functions

The Trusted RUBIX Security Markup Language (RXSML) provides a variety of functions to manipulate attribute values. Functions exist to compare, transform, mathematically manipulate, perform set operations upon, and perform bag operations upon attribute values. Functions generally accept some number of typed arguments and produce a typed result. Functions may be nested and operate over type values or set of values.

The following table gives the set of functions that are supported by Trusted RUBIX RXSML.

Comparison	Mathematical	Conversion	Boolean	Sets
• equal • not-equal • greater-than • greater-than-or-equal • less-than • less-than-or-equal • time-in-range • dnsName-match • ipAddress-match • regexp-match • MAC-check	• add • sum • multiply • subtract • mod • divide • round • abs • floor	• cast • string-normalize-to-lower-case • concatenate • string-normalize-space map	• or • and • n-of • not	• one-and-only • bag-size • is-in • bag • any-of • all-of • any-of-any • all-of-any • any-of-all • all-of-all • map • intersection • union • at-least-one-member-of • subset • set-equals

Policy Target

A **Target** defines the subjects, objects, operations, and times for which a Policy, Policy Set, or Rule is applicable. Specifically, it defines a set of subjects, resources, actions, and environments for which the parent Policy Set, Policy, or Rule element will be used in fulfilling a decision request. If the current context does not match the target then the element will not be used in satisfying the decision request.

The Target is used to define subjects, resources, actions, and environment that may be easily indexed. This allows indexes to be built that provide fast matching between a decision request and associated policy. The method of specifying the matching set of attributes is therefore restricted to simplistic comparisons. The set of functions that may be used within a Target are known as matching functions and are a subset of the total set of functions provided by the RXSML language. Valid matching functions are:

- equal
- not-equal
- greater-than
- greater-than-or-equal
- less-than
- less-than-or-equal
- regexp-match
- dnsName-match
- ipAddress-match

Policy Rule

The rule is the most elemental construct that may produce a decision outcome. A Rule element may evaluate to Permit, Deny, Not Applicable, or Indeterminate and may exist only within a Policy element. The main components of the Rule element are the *target*, *effect*, and *condition*.

The **Target** element of a Rule defines the context for which the rule applies. Specifically, it defines a set of subjects, resources, actions, and environments for which the rule will be considered in calculating the decision outcome of the parent Policy. If the current context does not match the Target then the rule will not be used in reaching the decision outcome.

The **Effect** is an XML attribute of the Rule element and defines the outcome (Permit or Deny) for the Rule if the Condition element evaluates to TRUE.

The **Condition** element contains a predicate that represents the logic of the Rule. The Condition may evaluate to TRUE, FALSE, or Indeterminate. The outcome of the Condition controls if the Rule evaluates to its Effect (if Condition is TRUE) or to Not Applicable (if Condition is FALSE).

Policy Obligations

In addition to RXSML policies being used to control access to objects, they may be used to perform actions based upon the policy outcome. These actions are called **obligations**. Whether an obligation's action is executed is dependent upon the outcome of the policy and the configuration of the obligation. An obligation may be configured to execute on an outcome of Permit or Deny. Generally, if the policy containing the obligation evaluates to the specified outcome then the obligation's action is executed. Trusted RUBIX RXSML supports three obligations:

The **set-field** Obligation: Set the value of a row field during a row SELECT, INSERT, OR UPDATE operation. During the SELECT operation the set field will be viewed by the user performing the SELECT and no change is made to the row on disk. During the INSERT and UPDATE operations the set field will be stored on disk. The field may be set from a literal value or from a calculated value, using any RXSML function or attribute.

The **set-error-code** Obligation: Set the error code seen by the user performing an operation. This is useful for hiding the existence of an object. For instance, it allows a "does not exist" error code to be returned instead of an "access denied" error code, where the latter would reveal the existence of the object.

The **audit** Obligation: Write a custom audit record to the Trusted RUBIX audit trail. The audit record will always include a base set of information. It may optionally include any literal or calculated value, using any RXSML function or attribute.

For more information on the Trusted RUBIX ABAC and its use of obligations, please see the Trusted RUBIX Security Policy Manager Reference Guide.

Example: Cross Domain Releasability

This example demonstrates policy which releases information across security domains using very specific requirements. In this example we have two distinct security domains, *domain1* and *domain2*. We will use the OS-MAC MLS policy to provide basic separation between *domain1* and *domain2* and use the Trusted RUBIX ABAC to write highly specific rules to allow controlled information flows between the two domains.

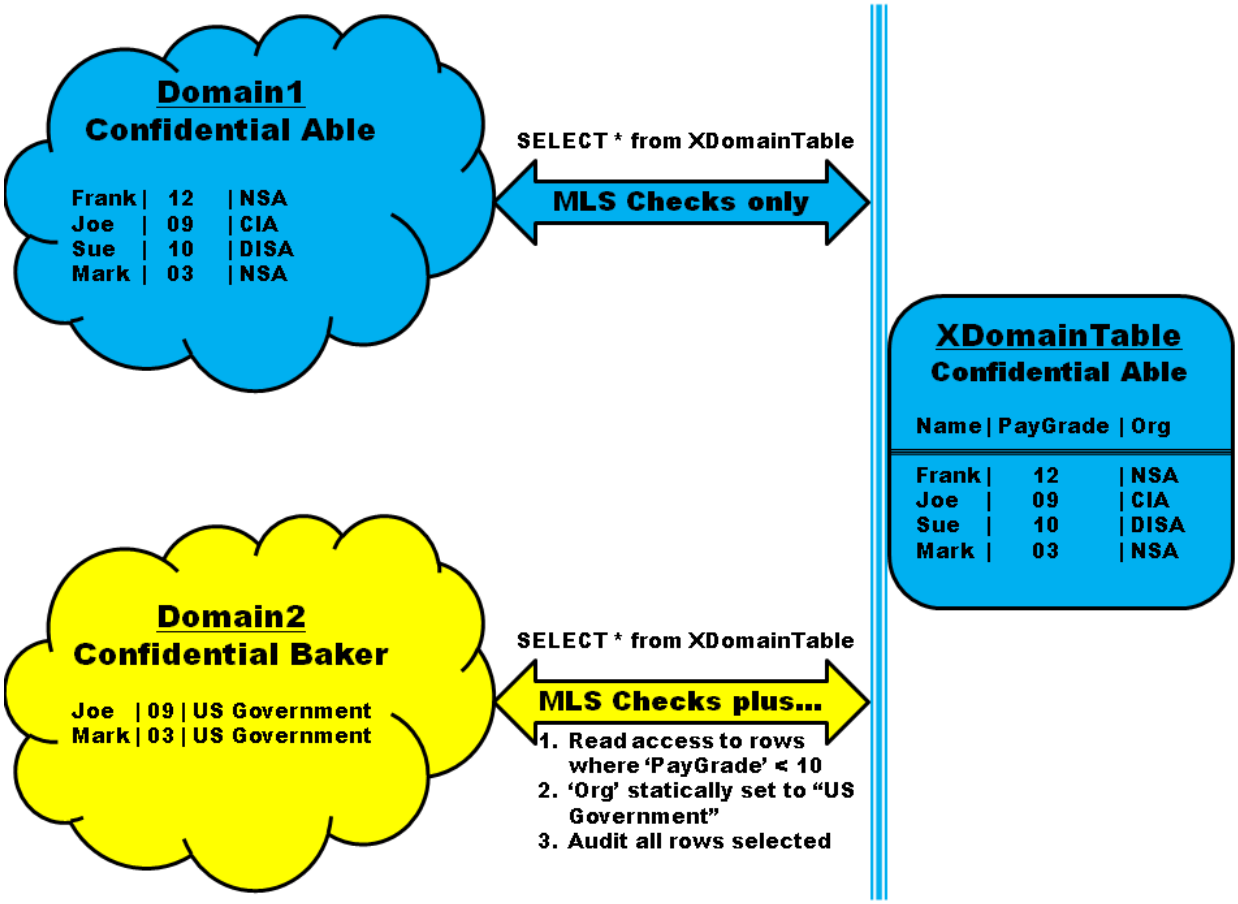
To see a detailed description of this example, please see the Trusted RUBIX Security Policy Manager Tutorial.

The security requirements are:

- Two domains: *domain1* and *domain2*
- *domain1* statically connected from the '*Confidential Able*' label
- *domain2* statically connected from the '*Confidential Baker*' label
- Single table (*XDomainTable*) in *domain1* containing information regarding US government employees with three columns: *Name*, *PayGrade*, and *Org*
- Read-write access by *domain1* to all rows of table
- Read-only access by *domain2* to a subset of rows in the table where *PayGrade* is less than ten
- Users from *domain2* must not see actual values of *Org* (e.g., NSA, CIA) but shall see a "cover" of 'US Government'
- The *Name* field of rows that *domain2* sees must be audited
- No access by *domain2* to remainder of rows in the table where *PayGrade* is greater than or equal to 10

The following diagram shows the architecture of the solution along with the results of both domains issuing a SELECT query. Note that *domain1* sees all of the rows unaltered while *domain2* sees only a subset of the rows with the *Org* field statically set to a literal value (US Government). Also note that row data is audited only when the policy permits access by *domain2* to a row.

Figure 5: Cross Domain Releasability



The RXSML security policy code is organized into four policies (*xdomain-select*, *mac_check*, *deny*, and

xdomain-open) and two policy sets (*table-obj* and *open-obj*). The following diagram shows the policies and policy sets and how they interrelate. Note that a policy may be used within multiple policy sets allowing modular policies and reusable code. Though not shown, a policy set may also be used within multiple other policy sets.

Policy and Policy Set descriptions and code are:

The ***mac-check*** Policy: Performs an OS-MAC security policy check on the current operation, evaluating to Permit if the operating system's MAC policy would allow the operation. The RXSML language provides a single function called MAC-check to query the OS-MAC policy. The function will return TRUE if the operating system's MAC policy would allow the operation and FALSE otherwise.

The ***deny*** Policy: Always denies the operation. Used as the "catch-all" policy within policy sets when no other policy within the policy set permits an operation.

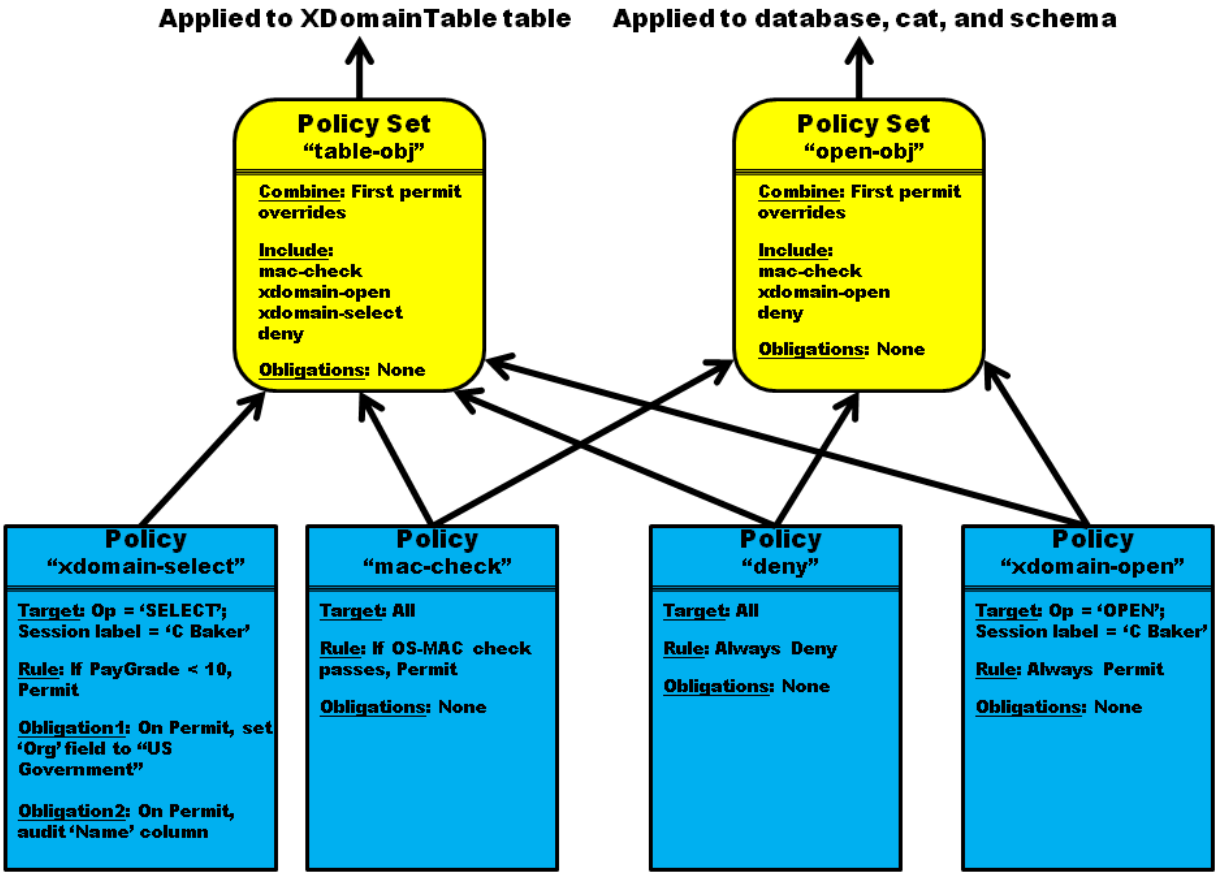
The ***xdomain-open*** Policy: Allows users from *domain2* to open DBMS objects that were created within *domain1*. By design, this policy supersedes the underlying OS-MAC policy. Note that this policy is targeted to OPEN operations from subjects at the '*C Baker*' session label (i.e., *domain2* subjects). It will have no effect on other operations.

The ***xdomain-select*** Policy: This policy contains the bulk of the cross domain logic. It targets only SELECT operations from subjects at the '*C Baker*' session label (i.e., *domain2* subjects). If the value of the *PayGrade* field of the row being selected is less-than 10, the operation is permitted. Additionally, if the operation is permitted, the *Org* field of the row being read is set to 'US Government' and the *Name* field of the row is audited, along with common audit information.

The ***open-obj*** Policy Set: This policy set contains all logic needed for *domain1* and *domain2* subjects to open *domain1* parent objects (database, catalog, and schema). The included policies are evaluated in order and the first Permit results in the policy set evaluating to Permit. If no included policy evaluates to Permit, then the deny policy will cause the policy set to evaluate to Deny. The *mac-check* policy will permit *domain1* subjects OPEN operations while the *xdomain-open* policy will permit OPEN operations for *domain2* subjects.

The ***table-obj*** Policy Set: This policy set contains all logic needed for *domain1* and *domain2* subjects to perform operations on the *XDomainTable* table. The included policies are evaluated in order and the first Permit results in the policy set evaluating to Permit. If no included policy evaluates to Permit, then the deny policy will cause the policy set to evaluate to Deny. The *mac-check* policy will permit *domain1* subjects table operations, the *xdomain-open* policy will permit OPEN operations for *domain2* subjects, and the *xdomain-select* policy will permit restricted SELECT operations from *domain2* subjects.

Figure 6: Cross Domain Releasability Policies



Example: Individual Tables with IP Address White Lists

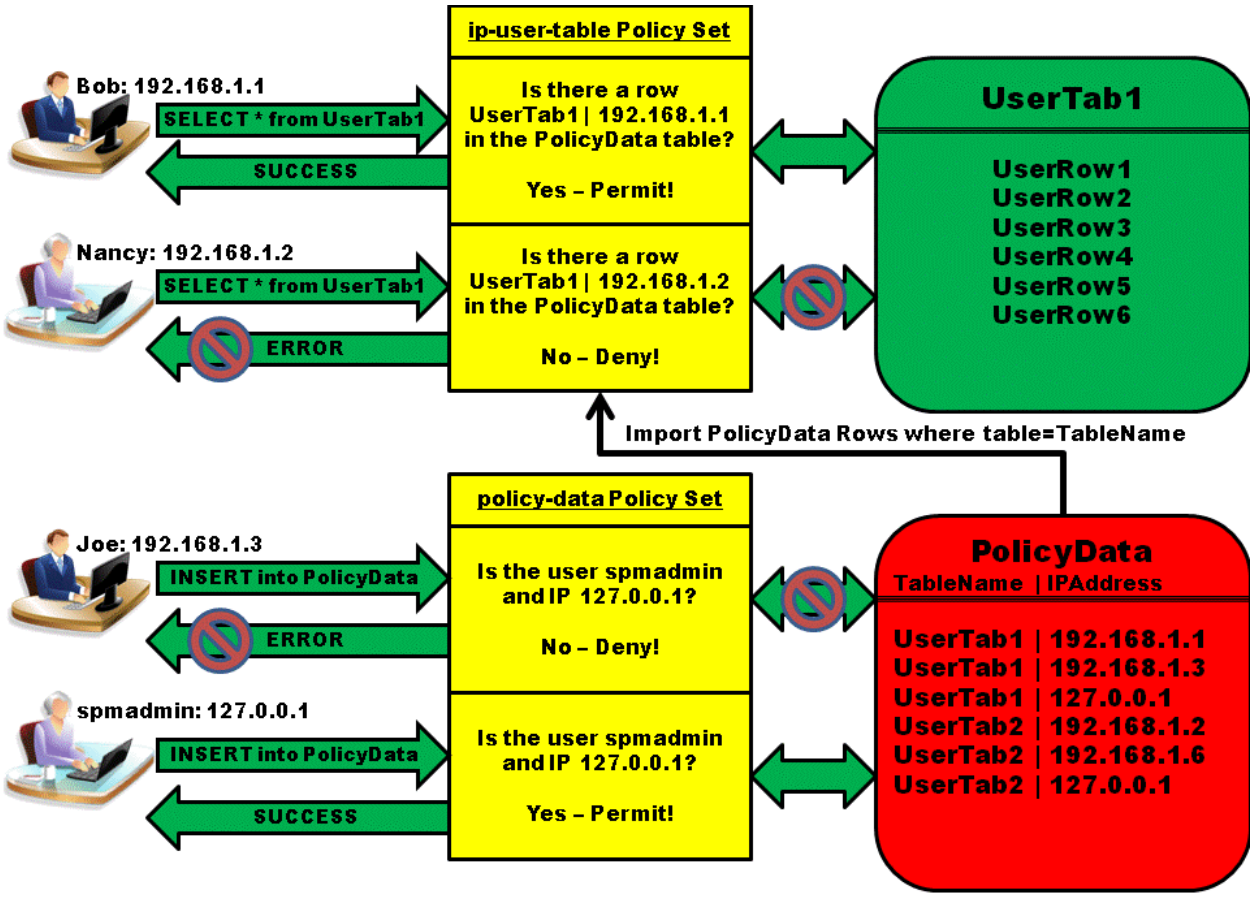
This example will use the Trusted RUBIX ABAC to create a white list of IP addresses for individual tables. The white list will map a named table to a set of IP addresses from which users are allowed to access the table. Furthermore, the white list will be maintained in a special table (*PolicyData* table) and will be updatable using standard SQL operations. Updates to the table containing the white list (*PolicyData*) are restricted to an administrative user (*spmadmin*) from the localhost (127.0.0.1).

To see a detailed description of this example, please see the Trusted RUBIX Security Policy Manager Tutorial.

The following diagram shows the architecture of the solution. There are two distinct security behaviors. The first restricts access to the *UserTab1* table only from permitted IP address as determined by the white-list contained in the *PolicyData* table. The second protects the security critical white-list information contained in the *PolicyData* table. The *PolicyData* table is only accessible by the *spmadmin* user connecting from the localhost (127.0.0.1).

This example demonstrates that the behavior of an ABAC policy may be dynamically configured in real time. This is accomplished by inserting, deleting, or updating rows in the *PolicyData* table.

Figure 7: IP White List Example



The RXSML security policy code is organized into one policy (deny) and two policy sets (*policy-data* and *ip-user-table*).

Policy and Policy Set descriptions and code are:

The ***deny*** Policy: Always denies the operation. Used as the "catch-all" policy within policy sets when no other policy within the policy set permits an operation.

The ***policy-data*** Policy Set: Because a DBMS table contains the IP address white-list information, accessing the information is restricted to a special user named *spmadmin* and only when he is connecting from localhost. The *policy-data* policy set performs this functionality. A policy is explicitly defined within the policy set (as opposed to being included) which targets users with a name of *spmadmin* and an IP of 127.0.0.1. When the policy target matches, all operations are permitted; otherwise, the included deny policy will deny the operation.

The *ip-user-table* Policy Set: Determines if the user's current IP address is allowed for the table being accessed. It imports rows from the *PolicyData* table, filtering them using the name of the table being access by the user. That is, all rows are removed where the *TableName* field does not equal the table being accessed by the user. It then compares the user's IP address with the resulting set of IP's contained in the *IPAddress* field. If the user's IP is found in the set of *IPAddress* fields then the operation is permitted; otherwise, the operation is denied.

Example: Row Access Restricted to Row Creator

This example demonstrates the use of the Trusted RUBIX ABAC to control access to the rows of a table such that only the creator of the row (i.e., the user performing the insert) will be able to access the row. This is accomplished through a special column in the table, called name, that holds the name of the user who inserted the row. Policy sets the value of the name column during insert and disallows any further update to the name column. Policy also only allows subsequent row operation to be performed on the row if the user name is equal to the value of the name column.

To see a detailed description of this example, please see the **Trusted RUBIX Security Policy Manager Tutorial**.

The following diagram shows the policy components and their relation to each other. The RXSML policy code is organized into four policies (*user-table-table-ops*, *user-table-sel-del-upd*, *user-table-insert*, and *deny*) and one policy set (*user-table*).

Policy and Policy Set descriptions and code are:

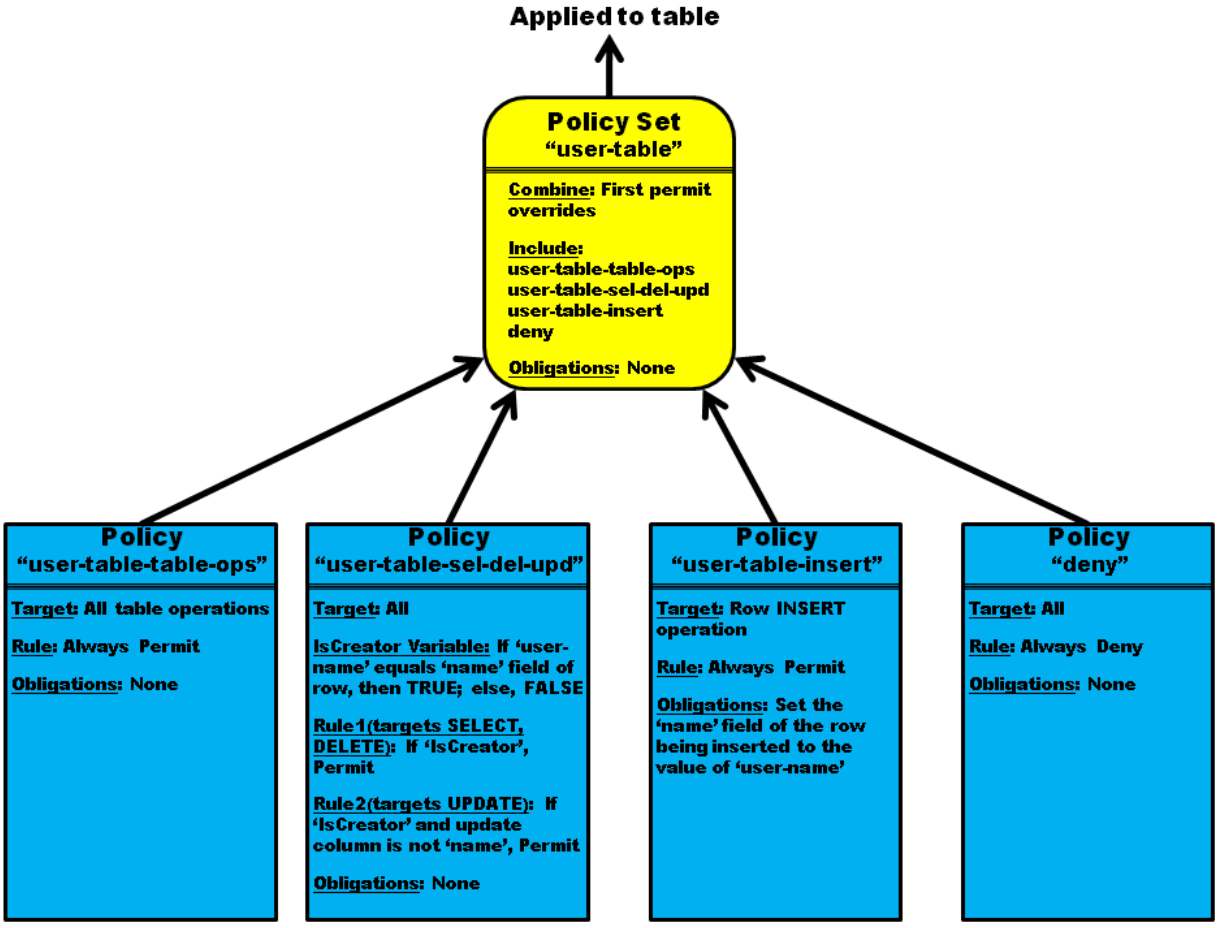
The ***deny*** Policy: Always denies the operation. Used as the "catch-all" policy within policy sets when no other policy within the policy set permits an operation.

The ***user-table-table-ops*** Policy: Simply permits all table operations.

The ***user-table-sel-del-upd*** Policy: Permits row SELECT and DELETE only if the name of the user performing the operation is equal to the value of the name field of the row being operated on. Permits a row UPDATE operation only if the name of the user performing the operation is equal to the value of the name field of the row being operated on and if the name field is not being updated. This policy demonstrates the use of the Variable Definition construct (*IsCreator*). This allows a named snippet of code logic to be defined and then referenced by name multiple times through the policy.

The ***user-table-insert*** Policy: Permits all INSERT operations and sets the name field of the row being inserted equal to the name of the user performing the insert.

Figure 8: Row Access Restricted to Creator - Policy



Appendix A: Frequently Asked Questions (FAQ)

1) In addition to DAC and traditional MAC is there another security method for controlling access to TR objects?

> TR provides three MAC security mechanisms; Multilevel Security (MLS), Type Enforcement (TE, SELinux only), and Attribute Based Access Control (ABAC).

2) What is ABAC and how is it enforced?

> Attribute Based Access Control (ABAC) is a comprehensive, generalized MAC policy security mechanism. It is characterized by modeling all of the information upon which one may wish to make a security decision as an attribute. It then provides a TR security enforcement rules engine (Security Policy Manager – SPM) to interpret highly customized security policies based upon the security attributes. Unlike other MAC mechanisms, ABAC need not be centered on the identity of the user. Instead, it may focus on an attribute the user has, such as the environment in which the user is operating (such as user Internet address or the data being accessed).

3) In TR how is ABAC implemented?

TR implements an ABAC security enforcement mechanism based upon the eXtensible Access Control Markup Language (XACML) standard from OASIS. XACML allows security administrators to write XML based security policies using a host of attributes from the TR environment. In addition, policies may be modular and nested in that the outcome of one policy may be an attribute for a different policy.

Policies may control access to every TR object and be updated and applied in "real time". If desired, the ABAC policy can fully coordinate with the underlying MLS security policy, allowing highly specific flows between security domains. The ABAC security mechanism of TR was created to address the needs of information sharing coalitions which require very specific policies to control release of information in "cross domain" environments.

4) What is the current status of the TR ABAC security policy mechanism?

The TR Security Policy Manager (SPM) is fully operational and supported using the TR "custom" XML Security Policy Markup Language (RXSML)". In its continuing efforts to minimize "proprietary" software whenever possible, TR is currently developing a version to be compliant with the XACML standard. In this manner, TR could integrate with other XACML standard aware applications and be controlled from a single administrative point. The ability to administer a wide range of applications from a single coherent source is attractive with Software as a Service and Cloud Computing.

5) Why are ABAC policies considered to be non-discretionary?

It is non-discretionary in that the owner of an object has no special abilities with regard to the policy behavior. Also, there are no authorizations that allow an administrative user to supersede the ABAC policy. If it is desired to give an administrative user special access, that logic would be written into the policy itself using the RXSML language. The ABAC policy also allows actions to be performed such as auditing based upon policy logic. The ABAC policy is configured by security administrators who create, edit, and assign security policies to objects using administrative tools. Policies may be changed in real time by a security administrator.

6) How do ABAC and TOS-MAC policies interact with each other?

An ABAC policy may be configured to override the TOS-MAC policy. In such a case the TOS-MAC policy is disabled for that particular operation and the ABAC is relied upon to control access. This allows policies to be constructed that allow information flows beyond those allowed by the TOS-MAC policy. These are known as "releasability" policies. A special function (MAC-check) is provided to calculate and duplicate the behavior of the TOS-MAC policy. If policy is not configured to override the TOS-MAC policy, ABAC may further restrict operations beyond the TOS-MAC policy. This is known as a "refining" policy and is the default policy behavior.

In this case, both the ABAC policy and the TOS-MAC policy must allow an operation for it to be permitted. During read based operations (e.g., object open and row selects) the ABAC policy operates upon the set of objects that have been filtered by the TOS-MAC. That is, objects that have been filtered by the TOS-MAC policy will not appear to exist when the ABAC policy is evaluated. If an object has no associated ABAC policy then the TOS-MAC policy is automatically enforced.

7) How are attributes used in creating ABAC policies?

The attributes used to write policy logic are typed (e.g., string, integer, etc.) and are categorized as subject attributes (e.g., subject name, subject IP address), resource attributes (e.g., object name, object label, row values), action attributes (e.g., operation, operation category), and environment attributes (e.g., system date and time). The functions used to manipulate attribute values are categorized as logic functions (e.g., and, or), comparative functions (e.g., equal, greater than), conversion functions (e.g., cast convert to lower case), group-of-values functions (e.g., testing if a value is in a group of values) and set functions (e.g., intersection, union).

8) Can a normal user (not a TR Security Administrator) relabel data using the SPM ?

While a normal user cannot alter the actual label of a row, given sufficient SPM permissions a normal user can copy data from a row with any row label into a row labeled with the user's session label. Additionally, if you have sufficient SPM permissions, the user could delete the original row. This has the same effect as reclassifying the row, but only to the user's session label. Note that specific ABAC policies would need to be applied to the database to permit such operations that supersede the MLS policy.

9) Can the SPM policy selectively "violate" or completely supersede the MLS and TE policy?

Yes, ABAC policy may be applied that allows a user to violate MLS or TE policy. The ABAC policy may allow the user to completely supersede the MLS or TE policy or it may allow it only for a specific operation and context. ABAC policy rules allow dominance checks to be performed to determine which cases are allowed.

10) Can the SPM policy be based on a single attribute?

Yes, it can be based upon a single attribute and that attribute can be the user ID. However, assigning policies to users is not a fundamental part of the SPM. That is, there is no requirement to have the user ID be part of an ABAC policy.

In other words, ABAC policy applies (or does not apply) to a specific instance of an operation based upon logic using many attributes, one of which **may** be current user. You can have any number of policies that target any number of users.

11) What are the two main security objectives of ABAC and the Application User mechanism?

A typical ABAC security policy used in conjunction with the Application User mechanism enforces the following security objectives. The first is to restrict row access to the Application User that created it. This will eliminate or drastically reduce SQL injection and URL manipulation attacks. The second objective is to deny the Application Administrator access to a database row unless the creating Application User is currently authenticated to the Application. This will greatly reduce the damage of an application hijacking attack.

12) Where can I find more information on the ABAC policy?

Only a simplified discussion of a typical ABAC security policy used in conjunction with the Application User mechanism has been addressed here. For a complete, working example, see the Security Policy Manager Tutorial. For detailed information about the TR Security Policy Markup Language (RXSML) used to create ABAC security policies, see the Security Policy Manager Reference Guide.

For details on the Application User mechanism, see the Application User Guide.