SOFTWARE ENGINE FOR XACML IMPLEMENTATION ON ROLE BASED ACCESS CONTROL

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Stephen O'Brien

SPRING
2014

SOFTWARE ENGINE FOR XACML IMPLEMENTATION ON

ROLE BASED ACCESS CONTROL

A Project

by

Stephen O'Brien

Approved by:

_____, Committee Chair
Ying Jin, Ph.D.

_____, Second Reader
Bill Mitchell, Ph.D.

_____
Date

Student: Stephen O'Brien

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

_____, Graduate Coordinator _____
Nikrouz Faroughi, Ph.D. Date

Department of Computer Science

Abstract

of

SOFTWARE ENGINE FOR XACML IMPLEMENTATION ON

ROLE BASED ACCESS CONTROL

by

Stephen O'Brien

This project is a continuation of a previous student's master's project, which produced a software engine capable of parsing through XACML policy files and used the collected data to produce role-based access control (RBAC) statements executed in Microsoft SQL Server [4]. However, there were a few important aspects of XACML that still required handling, including XACML request and response files, conflict resolution among policies, and policy refinement. This project handled these additional aspects, as well as performance evaluation, user interface updating, and code cleaning.

By making the additions stated above, this project has now fully realized its initial goal of parsing XACML policies into RBAC statements: a software engine has been produced that will completely automate the process of access control, needing nothing more than a set of policies in a simple XML format. Once a user has selected a directory containing the policies, the engine will sequentially select each file residing within, construct the RBAC structure, store the structure in a set of tables, resolve any conflicts arising from the structure, and produce the appropriate SQL commands to represent the structure

internally within the database. Additionally, if a user wants to determine if a database member has certain access permissions, the user can submit an XACML request. The engine will determine the result of the request and store it in an XACML response file. A user may also make changes to an existing XACML policy without worrying about conflicts with previously executed statements.

_____, Committee Chair
Ying Jin, Ph D.

_____
Date

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figures Page

Chapter 1

INTRODUCTION

XACML stands for eXtensible Access Control Markup Language [1]. It is a standard created by the OASIS group that uses XML to define policies for access control in database systems [1]. It is extensible in the sense that one can adapt or extend the language to fit one's individualized needs. Policies have a three-tiered hierarchical structure in XACML: policy sets, policies, and rules [2]. Each rule or collection of rules is a member of one policy, and each policy or collection of policies is a member of a policy set (see Figure 1). The purpose of having multiple rules or policies grouped together is to aid in organizing similar statements. For example, one policy could specify access to one table, or one policy could specify one type of access to multiple tables; in other words, a policy could grant SELECT and INSERT access to one table, or a policy could grant SELECT access to several different tables.

In the current version of the project implementation, due to its robust nature and widespread use in industry, Microsoft SQL Server 2012 Enterprise was the database of choice for interacting with, and testing the capabilities of, the XACML parsing engine [7]. Java was the coding language chosen when development of the parsing engine first started [9]. Additionally, Eclipse Kelper, Enterprise Edition, was chosen as the IDE (integrated development environment) because of it's Java integration technology and ease of use, such as the automatic library finder, which imports libraries based on method calls, and the built in debugger, which can step through the application line-by-line after

setting breakpoints [10]. Java was chosen for the wide array of libraries the enterprise edition, currently called J2EE, possesses. Namely, the Java Database Connection (JDBC) library was used to connect to the SQL Server database through localhost, and the XML parsing libraries located in javax were utilized to collect the data from the XACML files [6].

This project has been extended in the following ways. First, the engine can now parse request-formatted XACML files and produce the appropriate XACML response files. Second, the engine has an automatic conflict-resolver, which will check for conflicts among all rules in a policy set before they are converted into RBAC commands, and resolve them using the combining algorithms specified in the XACML files. Third, policies can be refined, or changed, at any time using a user interface for data input. Fourth, functionality was added to the engine to evaluate and provide printouts of its performance with the above tasks. Finally, the code was cleaned and heavily documented to support future work with the engine.

The rest of the project is organized as follows: Chapter 2 provides background information, both for XACML and the previous work that was accomplished on this project; Chapter 3 demonstrates the system implementation that went into completing the project, including request and response files, conflict resolution, policy refinement, and other engine enhancements; Chapter 4 delves into the process and results of the performance evaluation of the engine; and finally, Chapter 5 describes the summary of the project, future work, and a conclusion.

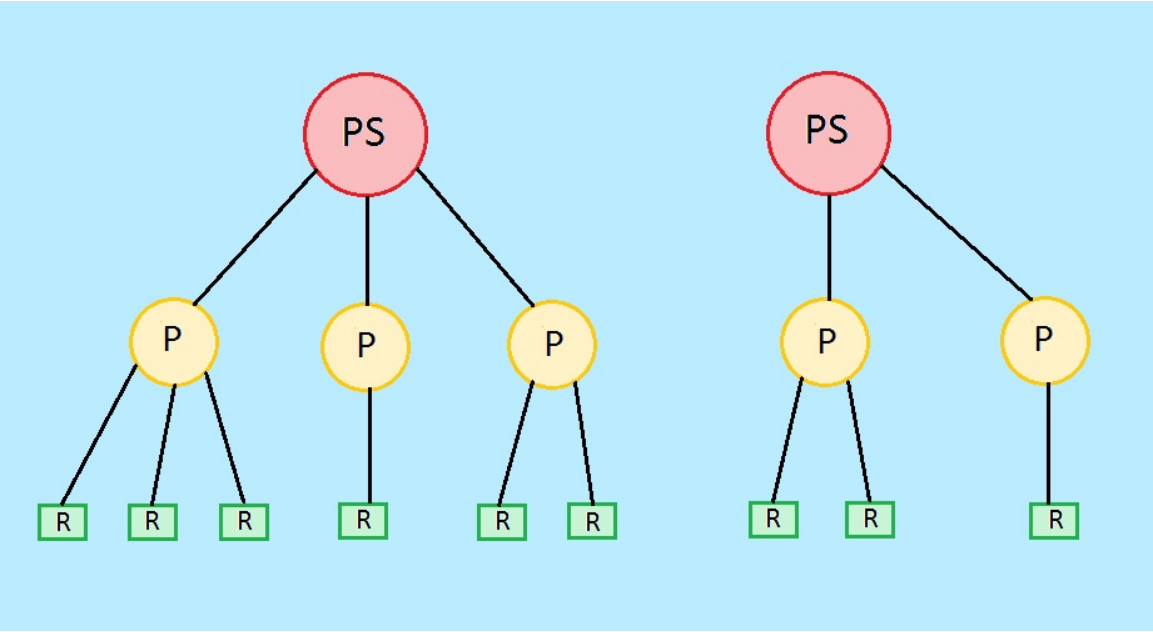

Figure 1 Policy Set Structure

Chapter 2

BACKGROUND

This chapter presents background of the XACML software engine, including alternate approaches for XACML use, and the previous work that was accomplished on this role-based access control approach.

2.1 Non-Database Approach

XACML is not limited to specifying access control on database systems, and may in fact be implemented on almost any type of computer system that requires access control policies. One such system in need of this approach is Social Networks on the Internet [5]. The authors of [5] inform of the biggest issue at the heart of access control on the Internet: privacy. With so many users sharing their personal information on these specialized websites, a standardized way to control other users' access to the content must be found. The authors then demonstrate how dynamic access control may be achieved using a combination of "sticky policies," and REL (Rights Expression Languages) in a distributed access control architecture [5]. Sticky policies are titled so because the policies travel with, or stick, to the data they are controlling [5]. The access control architecture is then based on XACML for standardization. Within social networks, users may change access permissions at a moments notice by choosing or changing which friends they want to share their data with. When the three aspects of

access control discussed are conjoined, they achieve the dynamicity required of social networking systems to handle those rapid access changes [5].

2.2 Access Control List Approach

Another research paper, MyABDAC (My Attribute Based Database Access Control), demonstrated the use of access control lists (ACLs) for implementing XACML policies in a MySQL database [3]. In this approach, the authors of the MyABDAC system compiled attribute-based policies into access control lists for an Oracle type database [8]. This system showed the benefits of using database-native support for XACML policies, specifically in the efficiency demonstrated by its performance evaluation [3]. This approach differs from ours in that it does not cover the role capabilities of XACML and all the benefits that are included. For example, in role hierarchy, one role can inherit permissions from another, and users with membership to multiple roles gain access to the associated permissions of all, each of which can dramatically reduce the number of access control entries needed in the database.

2.3 Previous Work

The conclusion of the research in [3] suggested that database systems are a very efficient way to implement XACML policies. It was thus decided by the authors of [4] to try a similar approach using role-based access control rather than access control lists. It was also decided to use J2EE as the platform for creating the engine that would handle the

parsing of the XACML policies, and the connection to the database for the execution of the produced SQL statements.

To parse through the files, two javax libraries were used: DocumentBuilder and DocumentBuilderFactory. Within these libraries are special methods to get XML attributes and elements, simply by specifying a tag or element name [4]. One can also use a Nodelist data type to pull a list of tags from the file, and then loop through the list to get individual elements from each tag. For example, the engine uses a Nodelist to gather all the policies from a policy set, then loops through the policies to create a Nodelist of the rules, and then loops through the rules to get the individual elements of each rule.

There are two different types of policy set in XACML: permission policy and role policy. The permission policy specifies access permissions, like “GRANT SELECT, INSERT ON code_table,” and associates them with a certain identifier [2]. A role policy has two functions: first, link database roles to permission policy identifiers; second, manage inheritance between roles by including one role in another role [2]. Once all of these connections have been made, the engine will produce the RBAC SQL statements (see Figure 2, where PPS stands for permission policy set, and RPS stands for role policy set). So, if a role policy were linked to the permission policy example above, with a role of “software_engineer,” the final RBAC SQL statements would be “GRANT SELECT ON [code_table] TO [software_engineer],” and “GRANT INSERT ON [code_table] TO [software_engineer].” Of course, there can be multiple rules in a permission policy, and multiple roles to link in a role policy. Figure 3 shows an example permission policy. First there is a “PolicySetId;” in this case the ID includes the name of the role it is intended for

to make it easier to remember. Next is the PolicyId, named in a similar fashion. Finally, the actual rules of the policy are listed. The first and only rule shown permits “SELECT,” “INSERT,” “DELETE,” and “UPDATE” on the “requirement_doc” table. Figure 4 shows an example role policy. The target specified in the policy is the “software_engineer” role. The “PolicySetIdReference,” which is used to link a role to a permission policy, is set to the same identifier seen in Figure 3. Figure 5 shows another piece of XACML: role assignment policies. These policies are not part of a policy set because they are only used to assign users to roles, and don’t need to be grouped in a special way. In the rule shown, the user “Bill” is being assigned to the “project_chief_manager” role. Again, there are usually many rules in a role assignment policy, each assigning a given user to a different role. Figures 3, 4, and 5 are modified from the report in [4].

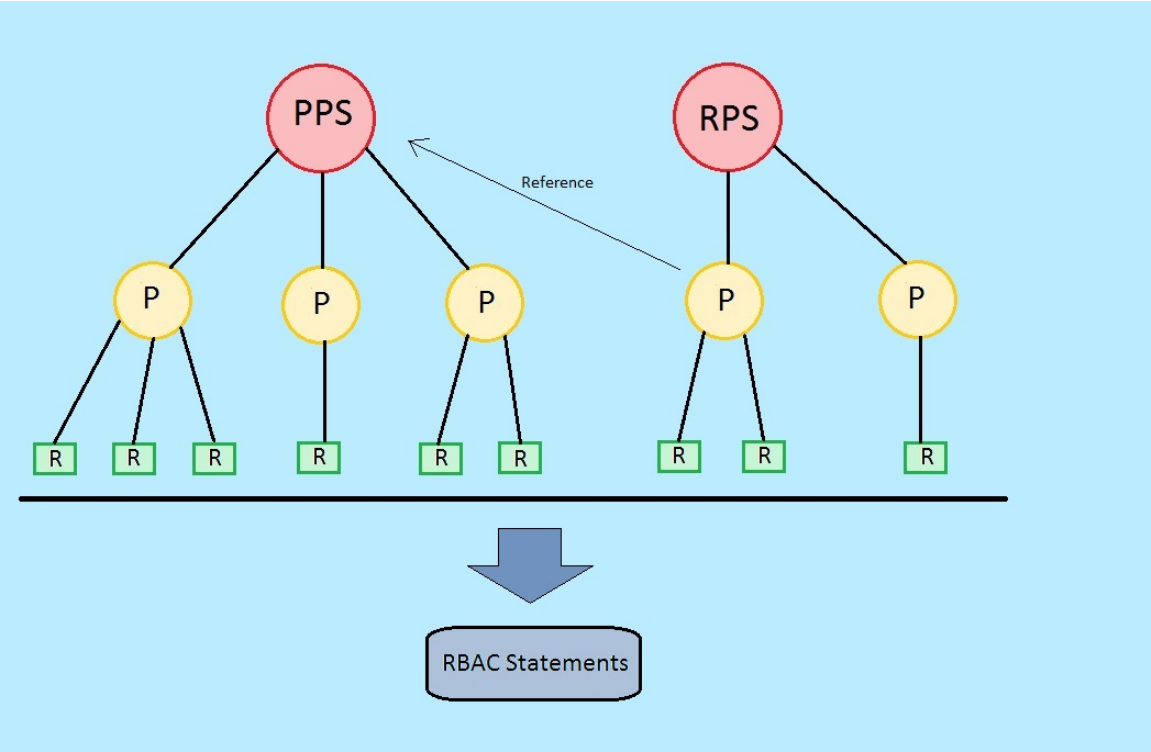

Figure 2 RBAC Generation Through Linking

```
PolicySetId="PPS:software_engineer:role"
Version="1.0"
PolicyCombiningAlgId="&policy-combine;permit-overrides">
  <Target/>

  <Policy PolicyId="Permissions:specifically:for:the:software_engineer:role"
  Version="1.0"
  RuleCombiningAlgId="&rule-combine;permit-overrides">
    <Target/>

    <!-- Permission to read and write table requirement_doc -->
    <Rule
    RuleId="Permission:to:read and write:table:requirement_doc"
    Effect="Permit">
      <Description>
        A software engineer may read or write on table requirement_doc
        only in projects assigned to him
      </Description>
      <Target>
        <Resource>
          <ResourceMatch MatchId="&function;string-equal">
            <Attribute
            AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
            DataType="&xml;string">
              <AttributeValue>
                requirement_doc
              </AttributeValue>
            </Attribute>
          </ResourceMatch>
        </Resource>
        <Action>
          <Attribute
          AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
          DataType="&xml;string">

            <AttributeValue>
              SELECT, INSERT, DELETE,UPDATE
            </AttributeValue>
          </Attribute>
```

Figure 3 Partial Permission Policy

```
<PolicySet xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17 xacml-core-
v3-schema-wd-17.xsd"
PolicySetId="RPS:software_engineer:role"
Version="1.0"
PolicyCombiningAlgId="&policy-combine;permit-overrides">
  <Target>
    <AnyOf>
      <AllOf>
        <Match
        MatchId="&function;anyURI-equal">
          <AttributeValue
          DataType="&xml;anyURI">
            &roles;software_engineer
          </AttributeValue>
          <AttributeDesignator
          MustBePresent="false"
          Category="&subject-category;access-subject"
          AttributeId="&role;"
          DataType="&xml;anyURI"/>
        </Match>
      </AllOf>
    </AnyOf>
  </Target>

  <!-- Use permissions associated with the software_engineer role -->
  <PolicySetIdReference>
    PPS:software_engineer:role
  </PolicySetIdReference>
</PolicySet>
```

Figure 4 Partial Role Policy

```
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17 xacml-
core-v3-schema-wd-17.xsd"
PolicyId="Role:Assignment:Policy"
Version="1.0"
RuleCombiningAlgId="&rule-combine;permit-overrides">
  <Target/>

  <!-- project_chief_manager role requirements rule -->
  <Rule RuleId="project_chief_manager:role:requirements" Effect="Permit">
    <Target>
      <AnyOf>
        <AllOf>
          <Match MatchId="&function;string-equal">
            <AttributeValue
            DataType="&xml;string">
              Bill
            </AttributeValue>
            <AttributeDesignator
            MustBePresent="false"
            Category="&subject-category;access-subject"
            AttributeId="&subject;subject-id"
            DataType="&xml;string"/>
          </Match>
        </AllOf>
      </AnyOf>
      <AnyOf>
        <AllOf>
          <Match MatchId="&function;anyURI-equal">
            <AttributeValue
            DataType="&xml;anyURI">
              &roles;project_chief_manager
            </AttributeValue>
```

Figure 5 Partial Role Assignment Policy

Chapter 3

SYSTEM DESIGN IMPLEMENTATION

This chapter presents my contribution to the software engine: support of XACML request and response files, a solution for conflicts arising from policies, an interface for handling policy refinement, and enhancements made to the existing engine.

3.1 Request Handler

An XACML request is designed to get a response to a question: does a particular object have specified access rights on another object [1]? The job of the request handler is to process this question, make a determination, and send back a properly formatted XACML response file. An XACML response is designed to simply answer the question with "Permit" or "Deny [1]." The request file is made up of a series of attributes: first, the user or role; second, the resource; last, one or more actions for access, like "SELECT", "INSERT", "DELETE", and/or "UPDATE." Figure 6 is an example of one such request file. This request asks "does Ace have SELECT access on the code table?"

To process the request file, the DocumentBuilderFactory library was used once again. First, the engine will cycle through each attribute until every required type has been received, including a subject category, either a user or role, a subject, which is the name of the user or role, a resource, which is a table in our database implementation, and a set of actions, like "SELECT", "UPDATE," and so on. For a subject category of "user," the

Transact-SQL function “HAS_PERMS_BY_NAME()” is called to make the determination. Any SQL Server user can call this function to determine if they have certain access permissions [11]. Therefore, to utilize the function correctly, the engine, which is connected as a database super-user, precedes the function call with an “EXECUTE AS USER” command, followed by the parsed user’s name. For a subject category of “role,” a temporary user must be created first, and then assigned to the parsed role. After doing so, the “EXECUTE AS USER” and function call may be used again. To prevent taking unnecessary space, the temporary user is then dropped from the database.

The “HAS_PERMS_BY_NAME()” function returns either a 1 or a 0, conveying that the subject does or does not have the access requested, respectively. Figure 7 shows an example of an XACML response file. The file contains a decision of “Permit” or “Deny,” corresponding to the function’s return of a 1 or 0. If there was some type of error in the request file, the decision is set to “NA,” representing “no answer” or “not applicable;” if there is an error within the engine itself, the result is “Indeterminate.” Once this determination has been made, the engine will create the properly formatted response file with the embedded decision, and store it in the root directory where the engine is being run. These stored request files may also serve as a log of access request activity.

```
<Attributes
Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject">
  <Attribute IncludeInResult="false"
  AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id">
    <AttributeValue
    DataType="http://www.w3.org/2001/XMLSchema#string">
      ACE
    </AttributeValue>
  </Attribute>
</Attributes>

<Attributes
  Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource">
  <Attribute IncludeInResult="false"
  AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id">
    <AttributeValue
    DataType="http://www.w3.org/2001/XMLSchema#anyURI">
      table:code
    </AttributeValue>
  </Attribute>
</Attributes>

<Attributes
  Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action">
  <Attribute IncludeInResult="false"
  AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id">
    <AttributeValue
    DataType="http://www.w3.org/2001/XMLSchema#anyURI">
      SELECT
    </AttributeValue>
  </Attribute>
</Attributes>
```

Figure 6 Partial XACML Request

```
<?xml version="1.0" encoding="UTF-8"?>
<Response xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
    <Result>
        <Decision>Permit</Decision>
    </Result>
</Response>
```

Figure 7 XACML Response

3.2 Conflict Resolution

When creating a permission policy, and especially if amending or adding to one, conflicts may arise between the rules. There are two levels at which this may happen: first, the policy level, where rules within a single policy contradict one another; second, the policy set level, where rules from two different policies contradict one another. Figure 8 gives a visual representation of conflicts at different levels. Each policy set, and each policy within a policy set, has its own rule-combining algorithm, which is used to determine which rule should win in the case of conflict. The combining algorithm may differ between a policy set and the policies it contains; therefore, the algorithm for a policy is used to resolve conflicts first, and then the algorithm for the containing policy set is used to resolve conflicts from the remaining valid rules. The combining algorithms that are handled in the software engine include permit-overrides, where rules with "GRANT" have precedence; deny-overrides, where rules with "DENY" have precedence; and first-applicable, where the first rules processed have precedence [2]. The other XACML combining algorithm, "only-one-applicable," is not supported by our system; this algorithm requires all policies to be processed at once, and returns a result of "NA" if no policy is considered applicable, "Indeterminate" if more than one policy is considered applicable, and "Applicable," or valid, if only one policy is considered applicable [2]. It would only be used in highly specialized circumstances.

To resolve conflicts, tables were created in the database to store information from all policy sets, policies, and rules. There is a table for each type. The "Policy_Sets" table stores the policy set ID and combining algorithm. The "Policies" table stores the policy

ID and combining algorithm. The “Policy_Rules” table stores the rule information, including user or role, resource, action, effect, result, new, and executed. The “resource” field stores the table name; “action” stores one particular action the user can take, like “SELECT;” “effect” stores access control, like “GRANT;” “result” is used to determine if this rule should be used when generating the RBAC SQL statements, set to 1 initially, meaning yes; “new” is used by the conflict resolver for determining which rules to search for conflicts, set to 1 initially, meaning include this rule in the search; and “executed” informs the engine if a rule has already been converted to SQL and executed, set to 0 initially, meaning this rule hasn’t been executed yet.

Each table also has a primary key, and the three tables are linked through foreign keys in the “Policies” and “Policy_Rules” tables. To determine if a conflict exists, the “Policy_Rules” table is first JOINed with the “Policies” table to get the first combining algorithm to use. The engine then runs SQL commands that check if the user or role, resource, and action are the same, but the effect is different. In other words, one rule says “GRANT” and another says “DENY,” but everything else is the same. In this case, if the combining-algorithm is permit-overrides, the rule with an effect of “GRANT” is used, and the result of the other rule is then set to 0. Having a result of 0 indicates that this rule should not be converted into a SQL statement. Next, all *three* tables are JOINed and the combining algorithm of the “Policy_Sets” table is used to resolve conflicts. However, only the rules that win in the first round of conflict resolution, that is, the rules with a “result” of 1, are considered for the second round. Once all conflicts are resolved, the engine pulls all the rules from the database where the “result” and “new” fields are equal

to 1, and converts them into RBAC SQL statements. Once the user chooses to execute these statements, the "new" field is set to 0, and the "executed" field is set to 1. Setting the "new" field to 0 is done to prevent previously parsed rules from being executed more than once. Setting the "executed" field to 1 is done to determine which rules should be revoked if a conflict arises later, which will be discussed in a future section.

To handle automatic conflict resolution, the conflict resolver must run every time the software engine parses new policies. So, when a user enters the location of files to process, the engine will check in the database to see if any of the rules within the policies have been processed before. If so, a message indicating, "one or more rules from this policy have previously been processed," is printed and that policy is ignored. If not, the policy is added to the database. Once all rules from the new policies have been processed, the SQL builder is called. Within the SQL builder, before actually generating the RBAC SQL commands, the conflict resolver is called to check the rules in the database, and only the rules with "result" and "new" fields equal to 1 are selected. By automatically handling conflict resolution within the SQL builder, users never have to worry about checking for or resolving conflicts manually. Now that the RBAC commands have been built, the user may print or execute them at any time, and the "new" and "executed" fields will not be changed until this happens. Figure 9 shows the "Policy_Rules" table after conflict resolution has been run. There are two rows in conflict, indicated by the rule table IDs 189 and 192. Because the parsed algorithm, now stored in the Policies table, was "permit-overrides," row 189, which has a "DENY" effect, lost, and so its "result" and "new" fields were set to 0.

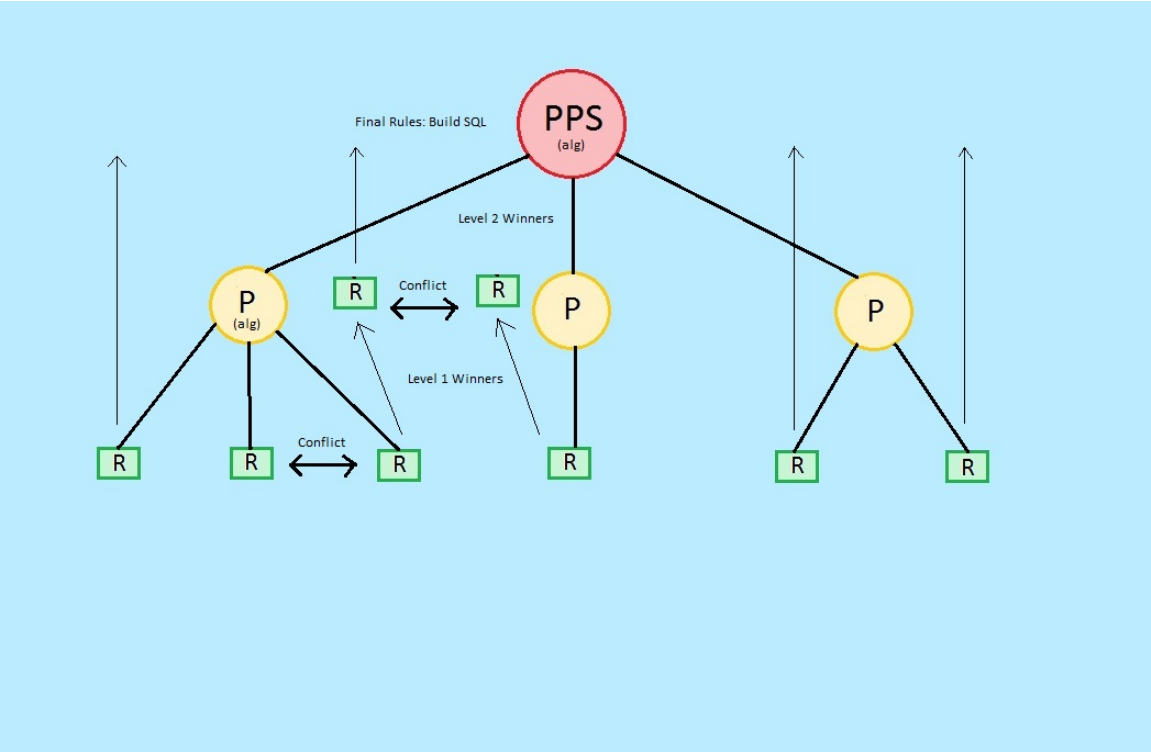

Figure 8 Levels of Conflict in Permission Policies

rule_table_id	rule_id	policy_table_id	user_role	resource	action	effect	result	new	executed
180	Permission:to:r...	1019	software_engin...	requirement_d...	SELECT	GRANT	1	1	0
181	Permission:to:r...	1019	software_engin...	requirement_d...	INSERT	GRANT	1	1	0
182	Denial:to:read a...	1019	software_engin...	requirement_d...	DELETE	DENY	1	1	0
183	Denial:to:read a...	1019	software_engin...	requirement_d...	UPDATE	DENY	1	1	0
184	Permission:to:r...	1019	software_engin...	code	SELECT	GRANT	1	1	0
185	Permission:to:r...	1019	software_engin...	design_doc ...	SELECT	GRANT	1	1	0
186	Denial:to:read:t...	1019	software_engin...	test_case_script...	SELECT	DENY	1	1	0
187	Permission:to:r...	1019	software_engin...	test_log	SELECT	GRANT	1	1	0
188	Permission:to:r...	1019	software_engin...	project_plan ...	SELECT	GRANT	1	1	0
189	Denial:to:read:t...	1019	software_engin...	project_plan ...	INSERT	DENY	0	0	0
190	Permission:to:r...	1019	software_engin...	project_plan ...	DELETE	GRANT	1	1	0
191	Permission:to:r...	1019	software_engin...	project_plan ...	UPDATE	GRANT	1	1	0
192	Permission:to:r...	1019	software_engin...	project_plan ...	INSERT	GRANT	1	1	0
NULL	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

Figure 9 Policy Rules Table After Conflict Resolution

3.3 Policy Refinement

Policy refinement is the act of inserting, removing, or updating XACML policies. One should be able to change anything from an entire policy set to an individual rule. To handle inserting, the user interface (UI) asks for all necessary IDs, starting with the policy set ID, then the Policy ID. Next, the IDs are checked for accuracy. If the policy set ID does not exist in the database, an error is printed. If the policy ID does not exist in the database, the combining algorithm is requested, and then it is inserted into the database. Once the first two IDs have been checked for accuracy, the UI will request the rule ID and the parameters of the rule to insert. After one rule is inserted, the UI asks if another rule should be added to this policy. If the answer is yes, the UI will request the next rule. If the answer is no, the UI will backtrack and ask if another policy should be entered within this policy set. This process repeats until all policies and rules the user desires have been inserted. Finally, the engine will return the user to the main menu. After inserting a rule, the policy set must be refreshed in the database. This is accomplished by setting the “new” and “result” fields for all rules within the policy set back to 1. Then the conflict resolver is run again to check for any new conflicts that may have arisen by inserting new rules.

Often when conflicts arise through this process, previously executed RBAC commands become invalid. Thus, another method was added to the conflict resolver to revoke these invalidated rules. Revocation is accomplished by searching for rules that have a “result” field set to 0, and an “executed” field set to 1, meaning a currently invalid rule has previously been executed. The engine performs a “SELECT” on this condition to pull

these rules out of the database, then builds new SQL commands by preceding the entire command with “REVOKE” instead of “GRANT” or “DENY”. Doing so removes the effect of the rule in the database. Once a rule has been revoked, the “executed” field is then set to 0 to prevent the engine from revoking the same rule multiple times. Figure 10 shows the process of inserting a new policy, indicated by the dotted line connecting it. After this policy is inserted, a new conflict arises between it and a previous policy within the same policy set. Because the old rule was already executed, a REVOKE statement is generated for that rule in SQL to remove its effect in the database, and the new rule’s effect take place instead.

The process for removal is similar to insertion. However, all IDs specified, from policy set, to policy, to rule, must already be contained in the database. If one is not, an error is printed and the user is returned to the main menu. Again, the user interface will allow the user to remove anything from an entire policy set to an individual rule. Any rule that is selected to be deleted, and has its executed field set to 1, will first be revoked the same way as insertion. However, in this instance the engine doesn’t need to search for rules to revoke, it simply revokes every rule the user chose to be removed. A new method was created to handle revocation for specific rules, with a parameter for the rule ID. Next, all rules under the policy set, policy, or rule IDs specified are deleted from the database. This process is simplified by setting the foreign key relationship between the tables to have a cascading effect. In other words, if the row with a specified policy set ID is deleted from the database, all associated data from the other two tables, connected to this ID, is also deleted from the database. Once again, the entire policy set is refreshed, and the conflict

resolver is run to handle any issues arising from the change in policy, but only if the policy set remains intact. When a user chooses an entire policy set for removal, no rules exist to check for conflicts, so no refreshing or conflict resolution is necessary.

The updating process can be simplified as the removal of a rule, followed by the insertion of a rule, where the rule ID is exactly the same for both steps. Therefore, the engine follows the steps for removal first, then follows the steps for insertion. However, the policy refresher and conflict resolver are not run until both the removal and insertion processes have completed.

Once the policy refiner has finished, and all conflicts have been resolved, the engine will automatically generate any new RBAC SQL statements and execute them, without requiring additional interaction from the user.

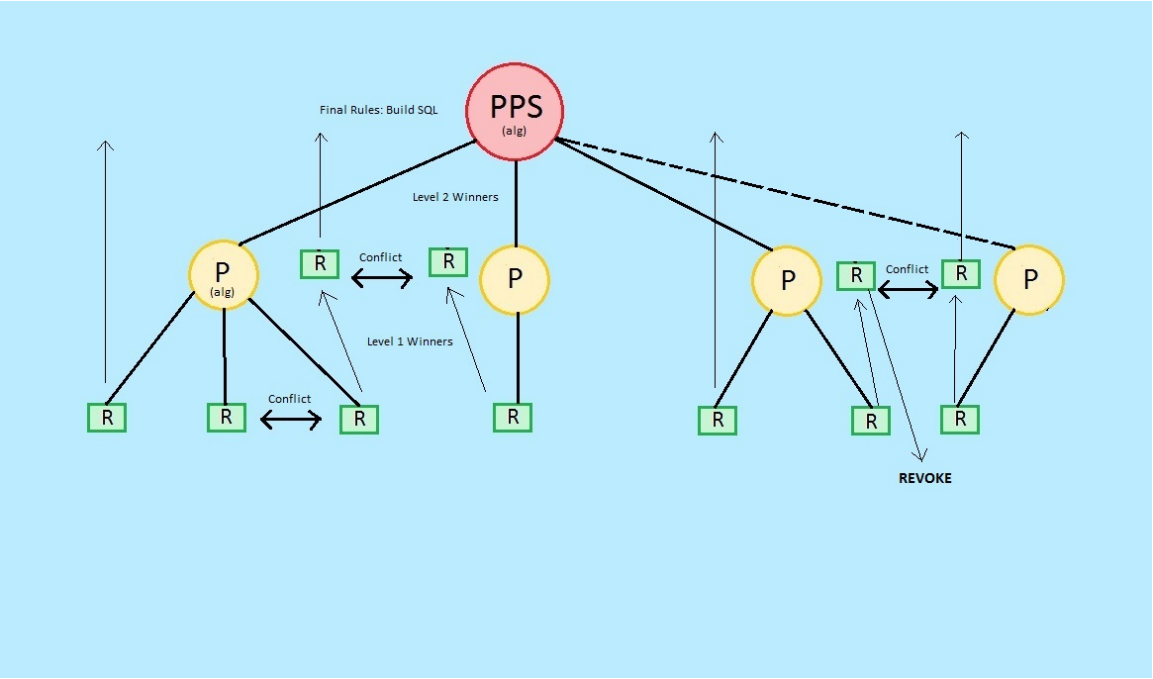

Figure 10 Conflict Resolution and Revocation After Refinement

3.4 Enhancements

The simplest of the enhancements made to the original program was to clean up the code and add comments to every method. This is important because there was difficulty in understanding a lot of the original code when the new work began. If another student takes over this project to handle the future work, specified later, it will be dramatically easier to do if each method has proper spacing, lined-up code blocks, and comments above and within to explain how the code is working. For example, there was an if-statement in the original code that contained several “&&”, and “||” characters, all with erratic spacing. By adding proper spacing and a few parentheses, it is much easier to understand what condition is being checked. Furthermore, by lining up code blocks, indicated by “{“ and “}” characters, it is easier to determine where one block ends and another begins, which again enhances understandability.

The next enhancement was to revamp the user interface. The first step was changing the keys being used to execute the available functions. For example, the original engine had a user type the “f” key to execute the produced SQL. However, it makes more sense to use the “e” key, as the word “execute” starts with an “e”. Next, the engine’s available functions were re-ordered to be more indicative of the order a user should take. For example, first a user should choose which files to upload, then the user should print the produced SQL, then the user should execute the SQL, and finally the user should refine any policies or quit. The UI now prints these functions in this order.

Another enhancement to the existing code was to allow users to specify an entire directory of XACML files, rather than specifying each file individually. This

dramatically reduces the number of steps a user must take to parse policies, and is especially useful for the performance section of this paper, where thousands of policy rules are parsed at once to generate performance metrics. If a user had to specify these thousands of policy rules with individual file names, it would take hours to accomplish. Now one only needs to specify a single directory that contains all the XACML files, and the engine will loop through each one on its own. Even if ten directories are used to separate the different types of policy files, the time needed to interact with the engine drops at least tenfold.

Finally, significant enhancements were made to accommodate the code for conflict resolution. First, instead of generating the RBAC SQL commands in the engine right away, the rules used for generation are first stored in tables in the database. Then, once the engine is ready to generate the RBAC SQL commands, after conflicts have been resolved, they are generated first by pulling the winning rules back from the database, and then proceeding as the engine did before. In order to store the rule information in the database, it was also necessary to increase the size of a few variables that were being used to store data pulled from the policy files. This allowed the engine to include information like policy set, policy, and rule IDs, combining algorithms, and differing effects. Consequently, the engine now supports policy rules that “DENY” access, whereas before every rule had to be “GRANT” only. Additionally, these changes enabled the engine to handle one policy set across multiple files, whereas before one policy set had to be contained to a single file.

Chapter 4

PERFORMANCE

This chapter presents the process and results for running a performance evaluation on the XACML software engine.

4.1 Process

In order to evaluate the performance of the engine, a second program was created to generate hundreds of XACML permission policy, role policy, and role assignment files, which in turn produced thousands of rules for each type of file. Once these files were generated, each file type was stored in its own directory; e.g., all permission policies were in one directory, role policies were in another directory, and all role assignment policies were in a third. Next, the software engine parsed through the permission and role policies by reading from one specified directory, then the other. In order to measure how long it took to parse these files and generate the rules, Java's "System.nanotime()" method was used [9]. The engine assigns this method call to two variables, one before rules are built, and one after. By subtracting the end time from the start time, the engine is able to print out the total execution time for file parsing and SQL generation. Next, the parsed files are executed, with the same method call before an after, and again, the engine prints the time, in this case for SQL execution. This process was repeated about twenty-five times for each type of engine function, and each time the data retrieved from these printouts was

manually inserted into an Excel spreadsheet. Once all the data was inserted, an average was taken from the spreadsheet and given a visual representation in graph form.

4.2 Results

As one can see in Figure 11, it only took about four seconds to create 3000 rules. SQL Server on the other hand took quite a bit longer to process the rules that were created, about twenty-four seconds for 3000 rules. The combined time for the software engine and SQL server is shown in Figure 12, about twenty-seven seconds for the entire process. The results for Figure 13 on the other hand are quite different. Both SQL Server and the software engine require about the same amount of time to process and execute the user-role assignments, approximately four seconds for 3000 rules. Figure 14 shows the combined time, only about eight seconds for 3000 rules. Therefore, the software engine was very consistent in creating rules for parsed files, but SQL Server has a much easier time executing role assignments than access-granting statements.

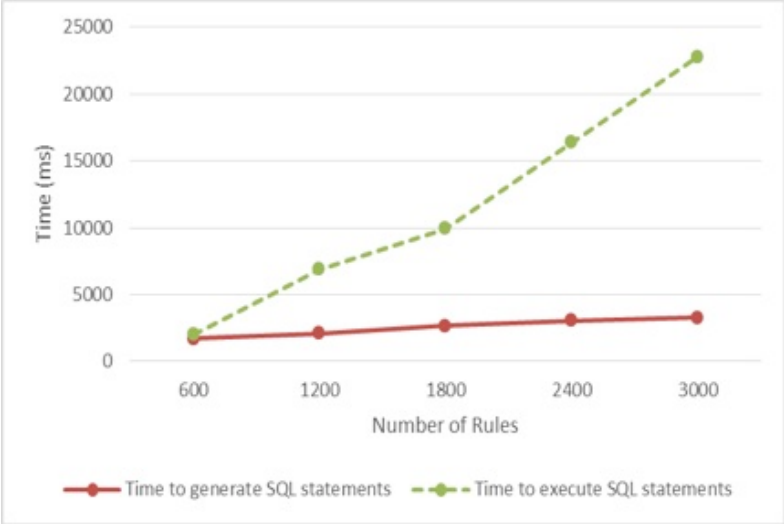

Figure 11 Separated Rule Generation and Execution Times

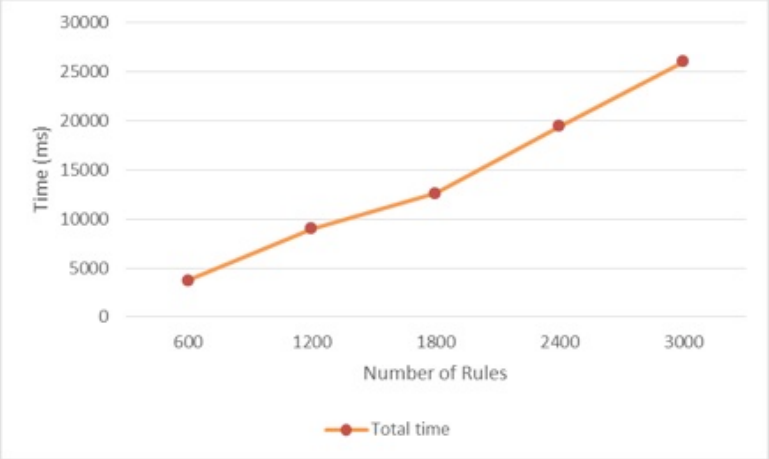

Figure 12 Combined Rule Generation and Execution Times

Figure 13 Separated User-Role Assignment Generation and Execution Times

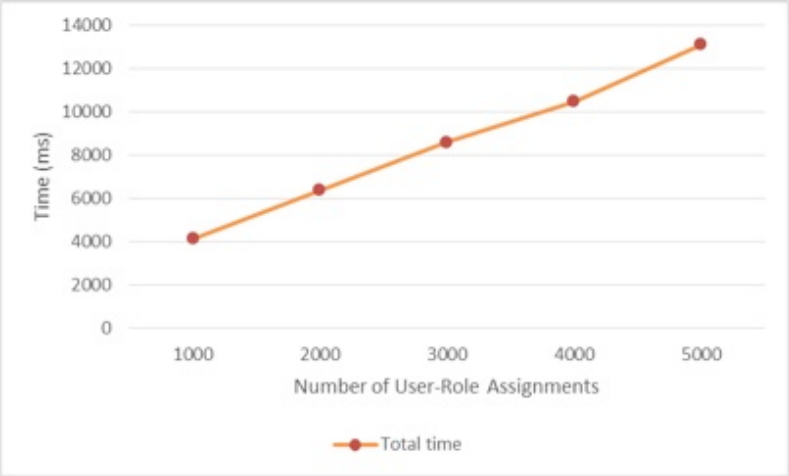

Figure 14 Combined User-Role Assignment Generation and Execution Times

Chapter 5

SUMMARY AND FUTURE WORK

This chapter presents a summary of the paper, future work that may be accomplished in the scope of this project, and a final conclusion.

5.1 Summary

This project demonstrated the continuation of a software engine from a previous project, capable of parsing XACML policy files and converting them into role-based access control statements for Microsoft SQL Server. Through improvements and additional functionality, including XACML request and response support, automatic conflict resolution within policy sets, policy refinement, and built-in performance evaluation, the functionality required of the original project idea has been fully realized.

5.2 Future Work

Much can still be done to improve the software engine. To start, the engine could be set up in conjunction with a web server, also coded in Java, to allow users to connect remotely. By doing so, the web server could serve HTML files, which would be used as an improved interface for manipulating the engine's functionality, and then send the user's data to the XACML processing engine. Currently, the user interface is presented

through a command-line interface only, and users cannot connect to the engine from a remote location.

A web interface brings about another improvement that should be made: security enhancements. Clients must be authenticated before being allowed access to such a powerful engine, with secure hashing mechanisms being used to store password information. The overall connection should also be encrypted using an SSL certificate that is trusted by the client process connecting to the server. Java includes libraries to make this easier, including the SSLServerSocketFactory library, which is used to establish an encrypted socket connection on port 443 [12].

Finally, separation-of-duty must be employed, both for security and organization purposes. Currently the tables that control policy, and the tables the policy control, are all being stored in a single database for connection simplicity within the program. Another database must be created specifically for the tables that control the policy, with a completely different database super-user and password for access. If a hacker were able to get access to the single database currently being used, both data and control would be compromised. Separating the data from the control creates a failsafe in such an event, and helps prevent confusion within the system.

5.3 Conclusion

The XACML software engine is a dynamic way implement role-based access control in a database environment. This is due to its always-on nature, where additional policies and role-assignments may be added at any time. With each new file added or policy refined,

the engine will automatically produce the complicated SQL required to make the policy work within a database system, handling all conflicts along the way. The only know-how required by the user is simple XML formatting of policies to enforce. In conclusion, this engine demonstrates our database-oriented approach to handle access control using XACML role-based access control policies.

Bibliography

[1] OASIS. “eXtensible Access Control Markup Language (XACML)" Version 3.0. Committee Specification 01, 10 August 2010. [online] http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf [Accessed: March 14, 2014]

[2] OASIS. "XACML v3.0 Core and Hierarchical Role Based Access Control (RBAC) Profile" Version 1.0. Committee Draft 03, 11 March 2010. [online] http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cs-01-en.pdf [Accessed: March 14, 2014]

[3] Jahid Sonia, Hoque Imranul, Gunter Carl A., Okhravi Hamed. “MyABDAC: Compiling XACML Policies for Attribute-Based Database Access Control.” CODASPY”11 Proceedings of the first ACM conference on Data and application security and privacy, New York, NY, USA, 2011, pages 97-108.

[4] Ying Jin, Travis Sorley, Jose Reyes. “A Database-Oriented Approach For XACML Implementation On Role-Based Access Control.” Masters Project. California State University, Sacramento. 2013.

[5] Anna Carreras, Eva Rodriguez, Jaime Delgado. “Using XACML for Access Control in Social Networks” W3C Workshop on Access Control Application Scenarios, 17 November 2009. [online] http://www.w3.org/2009/policy-ws/papers/Carreras.pdf [Accessed February 12, 2014]

[6] Oracle, (2014), Java Database Connectivity. [online] http://www.oracle.com/technetwork/java/javase/jdbc/index.html

[7] Microsoft, (2014), Microsoft SQL Server. [online] http://www.microsoft.com/en-us/sqlserver/default.aspx

[8] MySQL, (2014), MySQL. [online] http://www.mysql.com/

[9] Oracle, (2014), Java. [online] http://www.oracle.com/us/technologies/java/overview/index.html

[10] Eclipse, (2014), Eclipse IDE for Java EE Developers. [online] https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2

[11] Microsoft Developer Network, (2014), Has_Perms_By_Name (Transact-SQL). [online] http://msdn.microsoft.com/en-us/library/fooeaf8cc82-1047-4144-9e77-0e1095df6143.aspx

[12] Oracle, (2014), Class SSLServerSocketFactory. [online] http://docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLServerSocketFactory.html