A Graph Database Approach for XACML Role-Based Access Control Implementation

Arundhati Wahane and Ying Jin
Department of Computer Science, California State University, Sacramento
Sacramento, CA 95819-6021, USA
jiny@csus.edu

Abstract
Extensible access control markup language (XACML) is based on XML for defining fine-grained and coarse-grained authorization policies. XACML provides the mechanism for defining rules that are required to form authorization policy decisions. There are various approaches for XACML security policy implementation. This paper presents an approach to use a graph database to implement XACML role-based access control. Our system consists of two main components: XACML Policy Writer module and User Request/Response module. These modules have their own steps to define the functionality to handle XACML policy language and XACML request, managing and storing data in a graph database. It also handles complex hierarchical data structure and multiple relationships between nodes formed in the graph database. The paper presents our system architecture, design, and implementation using Neo4j.

Keywords: Graph database, Extensible Access Control Markup Language (XACML), Role-Based Access Control (RBAC)

1 Introduction

Nowadays, many real-time applications are dealing with lots of structured, semi-structured and unstructured data from different locations. The data may be terabytes or petabytes in size and may involve many interconnected data handling. The traditional relational database uses JOINs to query multiple tables. Joining tables, especially multiple large tables, are very costly, which may increase as dramatic data volumes increase.

NoSQL (Not only SQL) is an emerging approach for handling data-intensive applications. There are various categories of NoSQL databases, such as Key-Value store, Document store, Wide-Column store, and Graph databases. Graph databases deal with entities and relationships using nodes and arcs in a graph. It handles complex relationship between data more efficiently, as its top priority is around relationships between the entities. It avoids declaring referential constraints and does not need traditional JOIN operations because graph model allows the interconnection of nodes through arcs. The performance of graph database is relatively stable even with the increase of data size. Graph database is used in many different applications, such as fraud detection, real-time recommendation, identify and access managements.

XACML (extensible access control markup language) is one of the OASIS standard [7] to specify security policies independent of underlying implementation. In XACML role-based access control (RBAC), the permission to access data is based on hierarchical roles of the users. There are various approaches to implement XACML security policies. This paper describes our approach to implement XACML role-based access control policies using a graph database. We used the widely-used graph database Neo4j [4] as the underlying database system. The paper presents how to use graphs for complex hierarchical model to allow authorized users to access data. It also provided a solution to handle XACML request and response files through Cypher query language [5].

The rest of the paper is organized as follows. Section 2 presents XACML policy structures and related work. Section 3 describes the workflow of our approach. Section 4 describes system design and implementation. Section 5 concludes the paper with a summary of our approach.

2 Background and Related Work

Role-based access control (RBAC) uses the concept of “Role” and “Permission” to authorize the access to resources. A user is assigned to a particular role. A role is granted with resources.

XACML consists of policy languages to define policies and access control decision language for request and response. A policy set can have one or multiple policies. One policy can have one or multiple rules. The access control decision language allows a user to send a request with a question and receive a response to that question.
XACML role-based access control policies include: Role Policy Set (RPS), Permission Policy Set (PPS), and Role-Assignment Policy Set (RAP). Details can be found in [7]. Figure 1, based on the example in [2], shows an example of a permission policy on "requirement_doc" resource and the "SELECT, INSERT, DELETE, UPDATE" actions on that particular resource. The Role Assignment Policy Set describes the list of users, assigned to particular role.

Figure 1: Example of Permission Policy Set

XACML request file asks the question of whether a user can access the resource. The response file answers the question, which contains the access permission that is "Permit" or "Deny" on the resource.

There are different approaches for XACML implementation. In the implementation by SunXACML [9], policy verification is conducted at the application layer. XEngine [3] is also built to perform operations at the application layer. However, by using preprocessing, it improved performance. MyABDAC system [1] developed at University of Illinois Urbana-Champaign used MySQL relational database for processing XACML policies, using access control lists for attribute-based access control. Our past research, had applied relational database oriented approach for XACML implementation [2, 8], delegated authorization to the relational database role level.

3. Workflow

This section describes the workflow of our implementation system. In order to handle a XACML request, policies must be defined already that serves as the facts to grant or deny a request. Mainly, the system consists of two workflows, as shown in Figure 2 and Figure 3.

Figure 2 illustrates how to process a XACML policy and store the processing results. A policy writer produces a policy that is the input to our program called "XACML role-based access control application". Our program parses the policy and store it in the Neo4j graph database.

As shown in Figure 3, users can query the system by forming the XACML request file. This request file is an input to our program. The authorization of the query is implemented by performing queries to the graph database. The answer to their request file is sent back to users in the form of XACML response file.

To illustrate our approach, this paper uses an example of a Software Company application, which is based on the example in [2]. A Software Company has three level of hierarchy that consists of Chief Manager, Project Manager, and different types of Engineers, as shown in Figure 4. There are resources such as requirement documents, code, design documents, test case scripts, test log documents, project plan documents. In order to secure the access of resources, each role has different privileges when accessing the resources. These privileges are then indirectly available to their superior under whom the given role is working. The
permission types are select, delete, update and insert. For example, a design engineer has “select, delete, update, and insert” privilege to a resource named “design document”, while a test engineer only has “select” privilege to view the design document.

Figure 4: Role Hierarchy Structure

4. System design and implementation

Our implementation consists of two modules: 1) XACML Policy Module, 2) XACML Request/Response Module, as shown in Figure 5. The subsequent subsections describe each module in details.

Figure 5: Architectural Diagram

4.1 XACML Policy Module

The Policy Module is shown Figure 5 on the top part. The Policy Writer, who knows the semantic of the policies, produces XACML policies. Based on the policies, the Policy Module will process the policies and store them as nodes and relationships between the nodes in Neo4j. This module has two basic components: 1) Policy Parser Engine, and 2) CQL (Cypher Query Language) Interface, as detailed in the following subsections.

4.1.1 Policy Parser Engine

The Policy writer produces XACML policy files. There are separate files for Role Policy Set, Permission Policy Set, and User-Role Assignment Policy Set. The files serve as input to the Policy Parser Engine.

The policy parser engine extracts details of every block from XACML documents. The role of the user extracted by recognizing the “Policy Id” defined inside “Policy Set Id”. Each Policy Id’s has their own rule blocks to define what privileges specific role have on the resources and what role each user has to access those resources. The policy parser engine then collects details from “Policy Id” and stores the result of roles in role policy list.

The rule for User-Role Assignment Policy is to identify user in “Attribute Value” element related to “Subject” category. Then the user is assigned to specific roles when the match is obtained related to “Resource” category for each “Policy Id”. The policy parser engine collects the results of users in the role assignment list.

The rule for Permission Policy is to firstly identify the resources in “Resource” block. Later, “Action” block identifies what kind of access operation to perform on the resource. If the role has granted permission to utilize that resource, then the resource is assigned to that role. The policy parser engine collects the result of resources in the permission assignment list.

Permission Policy manages the hierarchies in the role structure when “Policy Set Id” refers to another “Policy Set Id”. The policy parser engine then assigns the junior role to corresponding senior role in the inherited role list.

4.1.2 CQL Interface

The output of the XACML policy parser engine sends the information to the CQL interface. The CQL interface component forms Cypher queries that are executable in Neo4j.

In Neo4j, we implement users, roles and resources as separate nodes. Firstly, starting by assigning a role to a user, and then the role can have a specific permission on each resource, as shown in Figure 6. The hierarchies between senior role and junior role nodes are by the specifications of parent-child relationship between them. Figure 6 shows the role-based access control model in Neo4j.

The system uses nodes to present users, roles, and resources respectively. The job of CQL Interface is to first create user, role and resource (object) nodes. This information obtained from the role assignment list, role policy list, and permission assignment list respectively as the output of XACML parser. The below query statements
show sample node creation using Cypher query language of Neo4j.

Query #1 Creating “user” node

CREATE(u:USER{id:"U1", name:"Bill"});

Query #2 Creating “role” node

CREATE(r:ROLE{id:"R1", name:"project_chief_manager"});

Query #3 Creating “resource (object)” node

CREATE(o:OBJECT{id:"O1", name:"project_plan"});

The next job of CQL interface is to create relationships between the nodes. Relationships describe how nodes are related, formed as arcs in a graph. There are three types of relationships in our graph: “User-Role”, “Role-Resource”, and “Senior role-Junior role”. The User-Role relationship describes every user is matched from role assignment list and every role is matched from role policy list. Thus, it creates a connection by assigning “Enabled” property between a user node and a role node. Example of creating user-role relationship is in Query 4. The output of Query 4 is in Figure 7.

Query #4 Creating “user-role” relationship

MATCH(u:USER{name:"Bill"}), (r:ROLE{name:"project_chief_manager"}) CREATE(u)-[m:MEMBER_OF{assignment:"enabled"}]->(r)

The Role-Resource relationship describes every role is matched from role policy list and every resource is matched from permission assignment list. The privileges that each role have on particular resource defined in the same permission assignment list. It creates a connection between a role and a resource by categorizing them into permission types and then granting permission property. Example of creating role-resource relationship is in Query 5. The output of Query 5 is in Figure 8.

Query#5 Creating “role-resource” relationship

MATCH(r:ROLE{name:"software_engineer"}), (o:OBJECT{name:"design_doc"}) CREATE(r)-[op:SELECT{permission:"permit"]}->{o}

The Senior Role-Junior Role relationship describes that a senior role inherits access rights from its junior role. There is a parent-child relationship to connect them by making relation property as “true.” It allows a senior role to access the resources granted to its junior roles. Example of cypher query senior role-junior role relationship is as follows. The output of Query 6 is in Figure 9.

Query#6 Creating “role-resource” relationship

MATCH(a:ROLE{name:"project_manager"}), (b:ROLE{name:"software_engineer"}) CREATE(a)-[p:PARENTOF{relation:true}]>({b)

The output of CQL Interface is passed to Neo4j through Neo4j-Java driver connection using Bolt protocol [6]. Figures 7-9 show the graph as the result of the above queries. Green color nodes are users; blue color nodes are roles; and red color nodes are resources. Figure 10 shows the User-Role-Resource assignment for a junior role with multiple users and multiple resources.
Figure 10: Software Engineer Role

Figure 11 shows the example of output obtained in Neo4j with users assigned to senior and junior roles along with resource assignments.

Figure 11: RBAC nodes and relations in Neo4j

4.2 XACML Request/Response Module

This module allows users to submit a XACML request and receive an answer on whether a given user can access a resource with particular privilege. There are four steps to handle a request. The first step is to load the XACML request file, which describes the name of the user, the resource, and the privilege. Figure 12 describes an example of XACML request file.

The second step is to parse the file to extract relevant information using Request Parser Engine. The third step is to form cypher query statement to allow Neo4j to understand the input query.

The request handling must also consider senior and junior role hierarchy when accessing resources. A senior role inherits the resources assigned to its junior roles. There are two approaches to handle the inheritance. One approach is to explicitly assign each resource to the senior role, which results in one more Role-Resource relationship for each inherited resource. In the graph, there will be one more arc created from the senior role to the resource directly. The second approach is not to implement this additional arc in the graph, instead, relay on graph traversal to handle the inheritance. We used the second approach. The main advantage of the second approach is that when a junior role is assigned to a different resource, the senior role is automatically granted permission to access the new resource. When a junior’s resources are revoked, the corresponding resources from the senior role will be revoked automatically, if the resources were inherited from the junior originally. This approach makes the structure of the graph simple, clear, and adaptive to changes, especially for multi-layer role hierarchies. Query 7 shows an example formed at the third step of query interface sub-module. Figure 13 shows query execution result of this query through Neo4j. In this query, “Ace” is assigned to a role. The role can either access “code” object directly, or indirectly through role hierarchies. Because of the specific structure of this graph, the traversal from USER to ROLE is restricted to the scope of using single directional “MEMBER_OF” relationship from the given user to a role and single directional “PARENT_OF” relationships from

Figure 12: Example of XACML Request file.
From junior roles to senior roles.

**Query #7 Request CQL**

```cql
MATCH p=(u:USER)-[*]->(r:ROLE)-[op:SELECT]->(o:OBJECT)
WHERE u.name="Ace" AND o.name="code"
RETURN op.permission, r.name
ORDER BY length(p) asc
```

The last step of this module is to collect the result from graph database to produce a new XML response file, which is then returned to the user of the system. The system looks for permission result, such as “permit” or “deny”. If “permit” is present, then the final output is “permit”. For example, as Query 7 illustrated, the return “op.permission” is “permit”, and then the response to the request is “permit”. The corresponding Response file is shown in Figure 14. Similarly, if the return of “op-permission” is “no changes/no records”, then the response to the request is “deny.”

![Querying the Request CQL command](image1)

**Figure 13:** Querying the Request CQL command

![Example of XACML Response file](image2)

**Figure 14:** Example of XACML Response file

### 5. Summary

This paper described our approach of using a graph database to implement XACML role-based access control policies. Specifically, the paper described how to process the XACML and store it in a graph. We also presented how to handle XACML requests based on the processed policies. Neo4j is the backend database for persistent storage and query processing. In addition, the problem of handling complicated queries of hierarchical role structure is also successfully handled. Our future work is to refine our implementation.

### Acknowledgement

This research is partially supported by Research and Creative Activity Faculty Award Program, California State University, Sacramento.

### References


