Attribute based Access Control for NoSQL Databases

Sumedh Kamble (212IS012)

Department of Computer Science and Engineering
National Institute of Technology Karnataka
Surathkal, Mangaluru, India

Abstract. NoSQL databases have recently gained popularity due to their capacity to efficiently manage large amounts of unstructured data. Role-based Access Control (RBAC) was employed in the majority of these databases. Permissions are associated with roles in role-based access control (RBAC), and users are made members of roles, thereby receiving the roles' permissions. However, RBAC has certain restrictions, such as the inability to create rules utilising unknown, the fact that permissions may only be applied to roles rather than objects, and so on. The high flexibility and dynamic nature of Attribute-Based Access Control (ABAC) has been frequently praised. An strategy for integrating ABAC into NoSQL databases, notably MongoDB, that normally only support Role-Based Access Control, is provided to overcome these limitations (RBAC). The implementation and performance results for ABAC in MongoDB are also reviewed, with the emphasis on how it can be used in other databases.

Keywords: NoSQL, RBAC, ABAC, MongoDb, Security and Privacy

1 Introduction

Access control is a basic element of the security infrastructure of any database. Every database security officer wants to apply the principle of less privilege, zero-trust, segregation of duties, and other best practices without harming the database workflow and creating computational overhead.There are several approaches to organizing an database access management system.Rule based Access control and Attribute are two most commonly used methods which are discussed below.
Role-based access control (RBAC) is an access control method based on defining user roles and corresponding privileges. The idea of this model is that every user is assigned a role. Every created role has a collection of permissions and restrictions imposed on it. An user can access objects and execute operations only if their role in the system has the relevant permissions.A user might be assigned to one or several roles. When a newuser is created, it's easy to assign a role to it. And whena user is deleted, there is need to change the role parameters or a central policy.

Attribute-based access control is a model that evolved from RBAC over time due to limitations of RBAC. This model is based on creating a set of attributes for any element of database. A central policy defines which combinations of user and object attributes are required to perform any action.The policies can use any type of attributes (user attributes, resource attributes, object, environment attributes etc).In contrast to role-based access control (RBAC), which creates predefined roles that carry a specific set of privileges associated with them and to which subjects are assigned, ABAC is the concept of policies that express a complex Boolean rulethat can evaluate to many different attributes. Attribute values can be set-valued or atomic-valued. Set-valued attributes contain more than one atomic value.

1.1 Problem Description

When it comes to successfully managing massive volumes of heterogeneous and unstructured data, NoSQL databases offer an advantage, which makes them perfect for a wide range of applications in use today, particularly online apps and Internet of Things applications that require tremendous scalability. When it comes to data security, however, such systems lag behind relational databases [2]. Effective access control is a critical security challenge. Due to its great flexibility and customizable data security ability at multiple degrees of granularity, Attribute-Based Access Control (ABAC) has begun to acquire favour for a wide range of applications [3]. The goal of this project is to solve the problem of integrating ABAC into NoSQL databases. We also go over how to add characteristics and dynamic assignments to MongoDB's underlying Role-Based Access Control (RBAC) framework. With no additional query overhead, the solution may be readily implemented into any MongoDB server.

1.2 Motivation

RBAC's motivation is to simplify management of authorizations.This necessitates such databases to have strict facts protection mechanisms.Attribute-Based Access Control (ABAC) has been broadly liked for its excessive flexibility and dynamic nature.Role Based Access Control (RBAC) is famous for ease of policy management, while Attribute Based Access Control (ABAC) is famend for bendy coverage specification and dynamic selection making capability.ABAC can offer dynamic, fine-grained and identity-agnostic get entry to manipulate, and is taken into consideration to be the subsequent technology of get entry to access control systems. With no NoSQL database presently offering support for ABAC and for this reason missing in security measures, they can gain from such systems.

1.3 Objectives

This paper focuses on addressing the challenge of integrating ABAC into NoSQL databases in particular focused on MongoDB since it is one of the most popular

NoSQL databases.To discuss an implementation enhancing MongoDB's underlying Role-Based Access Control (RBAC) system with attributes and dynamic assignments. The system created should be such as it can be integrated easily into any MongoDB server with no additional query overhead.

1.4 Organization of the Report

Rest of the paper is organized as follows: Section 2 discusses the exisiting work done in field of RBAC, ABAC and its integration with databases.Section 3 explains the proposed solution and its implementation details.Section 4 discusses the Experimental results and analysis done after the proposed approach is implemented.Section 5 concludes the paper.

2 Related Work

Extensions to RBAC by merging attributes and roles have been extensively studied.RB-RBAC model [5] use attributes automaticaly assign user role. Similar literature such as parameterized role [6, 7, 8], object sensitive role [9] and attributed role [10] are also proposed before.Also Several attribute based access control systems and models have been proposed. The UCON model proposed in [11] focuses on usage control where authorizations are based on the attributes of the involved components. It is attribute-based but, rather than depending on core ABAC concepts, it focuses on more advanced access control features such as mutable attributes, continuous enforcement.
In latest years, Attribute Based Access Control (ABAC) [13], [14] is getting an growing reputation for its ability to help relatively custom designed sorts of database protection. ABAC is a subsequent generation get right of entry to manipulate version that gives dynamic, context-conscious and risk-wise access manipulate. It overcomes the principle obstacles of the dominant access manipulate fashions (such as, RBAC [10]), by, on the identical time, encompassing their benefits.A few early studies efforts have focused the enhancement of NoSQL structures with superior access control features. However most otf these frameworks do now no longer help ABAC and aim a selected datastore only.Wang et al [12] proposes a framework that fashions a primarily attribute-based access manipulate machine the use of programming with set constraints of a computable set principle. This method specifically works to recognize on how set theory defines the created policy instead of model itself. Yuan and Tong [13] describe ABAC withinside the factors of authorization structure and coverage formulation. This method focus on enforcement degree in preference to policy level of the version.

3 The Proposed Approach for

The design and implementation of ABAC for MongoDB databases are covered in this section. Figure 1 depicts a summary of the system. The user and object

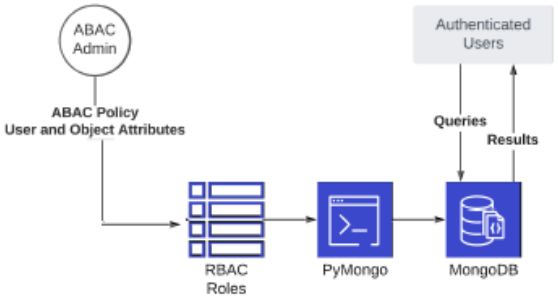

Fig. 1. Proposed Architecture

are provided by the ABAC system administrator. The ABAC policy, which is described in terms of an attribute values that are required for a specific action. As outlined below, there are a number of different roles that can be assigned. After that, the roles are formed in MongoDB The authorised users are then sent to MongoDB through PyMongo after the authenticated users can interface with the MongoDB database based on their permissions.

PyMongo is a Python module that allows you to communicate with a MongoDB database. This allows the algorithm described in [1] to be implemented in Python, and users may communicate with the database using PyMongo's interface to create new roles based on the mapping produced by the algorithm. As a result, it is possible to use MongoDB's RBAC system to create a functioning ABAC implementation.

The solution is compatible with all MongoDB drivers, servers, clients, and versions, and it makes MongoDB deployments simple. It communicates with the MongoDB server using basic PyMongo commands, which require database administrative access. PyMongo's compatibility with all versions of MongoDB is supplied by the MongoDB developers because it is the recommended manner of communicating with MongoDB from Python.

3.1 Conversion of ABAC policies to RBAC Roles

The User Attribute relation, the Object Attribute relation, and the ABAC Policy are all inputs to the Python class ABAC. The User Attribute relation is represented as a binary matrix, with a row for each user and a column for each attribute value that that user has chosen. Similarly, the Object Attribute relation is stored as a binary matrix, with each object having its own row. The ABAC policy is a tuple of User Attributes and Object Attributes that are necessary to provide a user with a list of rights for a certain object.

These are next processed for generating the set of authorizations using the algorithm described below. The time complexity of this algorithm is O(num_users x $num\_objects$ x num_rules x $num\_user\_attributes$ x $num\_object\_attributes$).

The algorithm is then used to determine the set of permissions for each user (User Permission Assignment) from the list of authorizations. This algorithm's time complexity is O(num_users x $num\_objects$).

There are also functions to alter and update the set of assignments, User Permission Assignment relations, and matching User and Permission Assignment relations whenever a new user, a new user, A new role or attribute is created or removed. This module's output can be regarded as a set of roles It can be used to dissect the specified ABAC policy.

User(u)	**EastCoast**(uc_1)	**Manager**(uc_2)	**Westcoast**(uc_3)	**Associate**(uc_4)
Alice	0	1	1	0
Bob	0	0	1	1
Mitch	1	1	0	0
sumedh	1	0	0	1

Table 1. UAR

Object(o)	**WestCoast**(oc_1)	**EastCoast** (oc_2)	**Customer**(oc_3)
sales	1	0	1
reports	0	1	1

Table 2. OAR

3.2 Algorithm

Step 1: Construct the set of Authorizations A from the User Attribute Relation (UAR), Object Attribute Relation (OAR) and the ABAC policy base (A).For each user(u_i)-object(o_j) combination from UAR and OAR, we check if their corresponding attribute conditions($U_C.u_i$ and $O_C.o_i$) form a superset of any of the given ABAC rules in A.

Step 2: Derive User Permission Assignment (UPA) from A: The UPA is defined as an M×N matrix, where $M = |U|$ comprising of a row for each user, and $N = |O - op|$, comprising of a column for each object and operation combination in A. Using (A), we derive (UPA) as follows: We consider all the Users in (A) and associate the objects with permissions to form $PRMS(o - op)$ in RBAC. There is a row in UPA for each user and a column for each $PRMS(o - op)$ For each row, if the $(o - op)$ is true for that user, the corresponding cell is filled with 1, otherwise with 0.

The output of this algorithm is the roles generated from ABAC policies which are then created in database and assigned to users.Above algorithm is implemented using Table.1 as UAR, Table.2 as OAR, Table.3 as $Policy(\pi_A)$, and generated Table 4. A and Table 5. UPA.The columns in UPA can be considered as roles generated from ABAC rules.

Attributes	**Permission**
uc_2,uc_3,oc_1,oc_3	$find$
uc_3,uc_4,oc_1,oc_3	$find$
uc_1,uc_2,oc_2,oc_3	$find$
uc_1,uc_4,oc_2,oc_3	$insert$

Table 3. $Policy(\pi_a)$

User $_u$	**Object** $_o$	**Permission** $_op$
Alice	sales	$find$
Bob	sales	$find$
Mitch	reports	$find$
sumedh	reports	$insert$

Table 4. $Authorization(A)$

User	**sales-find**	**reports-find**	**reports-insert**
Alice	1	0	0
bob	1	0	0
Mitch	0	1	0
sumedh	0	0	1

Table 5. UPA

3.3 Application in MongoDB & Code Snippets

To be able to establish and assign roles in the server, the Python software connects to the MongoDB server using Py-Mongo with an administration Role.If users are not present in databse they are created using *createUser* command of MongoDB and roles are assigned.The *createRole* command of MongoDB is used to create a role for each role in the Permission Assignment relation, and the *grantRolesToUser* command is used to assign users to each role based on the User Assignment relation. When the ABAC Policy is changed, or a new user or object is created, these procedures are called again based on the new User and Permission Assignment relationships.

```
#seaarch each policy in superset
satisfied_pol = []
satisfied_perms = []
for pol in policy['attributes']:
    #print(pol)
    if superset_attr.index(pol) != -1:
        satisfied_pol.append(pol)
        #policy[policy["attributes"]==pol].index.values)
#print(satisfied_perms)

#get user and object that satisfied policy
satisfied_user_attr = []
satisfied_object_attr = []
for i in range(0, len(satisfied_pol)):
    satisfied_user_attr.append(satisfied_pol[i].split("|")[0])
    satisfied_object_attr.append(satisfied_pol[i].split("|")[1])
#print(satisfied_object_attr)
```

Fig. 2. Function to find satisfied policies and its corresponding user and object

```
#find permissions on these each policy
final_perm_list = []
t1=[]
t2=[]

for i, rows in policy.iterrows():
    for p in satisfied_pol:
        if rows["attributes"] == p_:#and (p not in t1 or rows["permission"] not in final_perm_list):
            t1.append(p)
            final_perm_list.append(rows["permission"])
perms = pd.DataFrame()
perms["attr"]=t1
perms["permissions"]=final_perm_list
perms = perms.drop_duplicates(ignore_index=True)
final_perm_list = perms["permissions"]
```

Fig. 3. Function to find permissions on policy

```
def Create_User_Roles():
    UA = algorithm.Derive_User_Assignment()
    Roles = UA.columns
    Roles = Roles[1:]
    Users = UA["Users"]

    client = pm.MongoClient('localhost',
                            username='myUserAdmin',
                            password='admin',
                            authSource='admin')
    for role in Roles:
        obj = role.split("-")[0]
        action = role.split("-")[1]
        db = client[obj]
        roleName = _role+"_role"
        # coll = db.list_collection_names()
        # print(coll)
        present_roles = []
        for p_role in db.command("rolesInfo")["roles"]:
            present_roles.append(p_role["role"])

        if roleName not in present_roles:
            db.command("createRole", roleName,
                       privileges=[{"resource": {"db": obj, "collection": ""},
                                    "actions": [action]}],
                       roles=[])

    print("\n*** All ROLES CREATED SUCCESSFULLY ***")
```

Fig. 4. Function to create users with coresponding roles.

```
#now create users and assign appropriate roles to them
present_users = []
for i, row in UA.iterrows():
    for role in Roles:
        if row[role] == 1:
            #print(row["Users"]+"-"+str(role))
            obj = role.split("-")[0]
            db = client[obj]
            for u in db.command("usersInfo")["users"]:
                present_users.append(u["user"])
            if row["Users"] not in present_users:
                #print("creating user")
                db.command(
                    'createUser', row["Users"],
                    pwd=row["Users"],
                    roles=[{'role': role+"_role", 'db': obj}]
                )
            else:
                print("granting role to exisiting user")
                db.command({"grantRolesToUser": row["Users"],
                            "roles": [
                                {'role': role+"_role", 'db': obj}
                            ],
                            'writeConcern': {'w': "majority", 'wtimeout': 2000}
                            })

print("*** ALL USERS CREATED SUCCESSFULLY AND ROLES ARE ASSIGNED ***")
```

Fig. 5. Assigning roles to users.

```
> db.getRoles()
[
        {
                "_id" : "sales.sales-find_role",
                "role" : "sales-find_role",
                "db" : "sales",
                "roles" : [ ],
                "isBuiltin" : false,
                "inheritedRoles" : [ ]
        }
]
>
```

Fig. 6. Roles created in sales db

4 Experimental Results and Analysis

To test the above mentioned algorithm, firstly we have to enable Access Control in MongoDB(by default no Access Control is enabled). It can enabled by command : $mongod --auth$. After that we execute out first set of Policies for the above mentioned UAR and OAR. It then creates the users and corresponding roles as shown in Figure 6 and Figure 7. For eg. *Alice* user is created and is assigned $sales - find$ role. Figure 9.shows *Alice* can do find operation in *sales* database on *orders* collection. But if we try to *insert* something *Unauthorized Access Error* is returned. Now change the Policy to give Alice *insert* operation.After $sales - insert$ role is assigned to *Alice* ,*insert* operation is successfully executed as shown in Figure 8.

Fig. 7. Users created in sales db

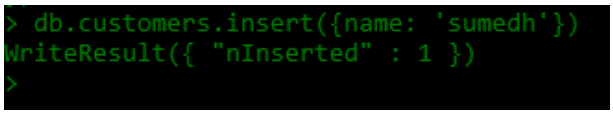

Fig. 8. After updating the role

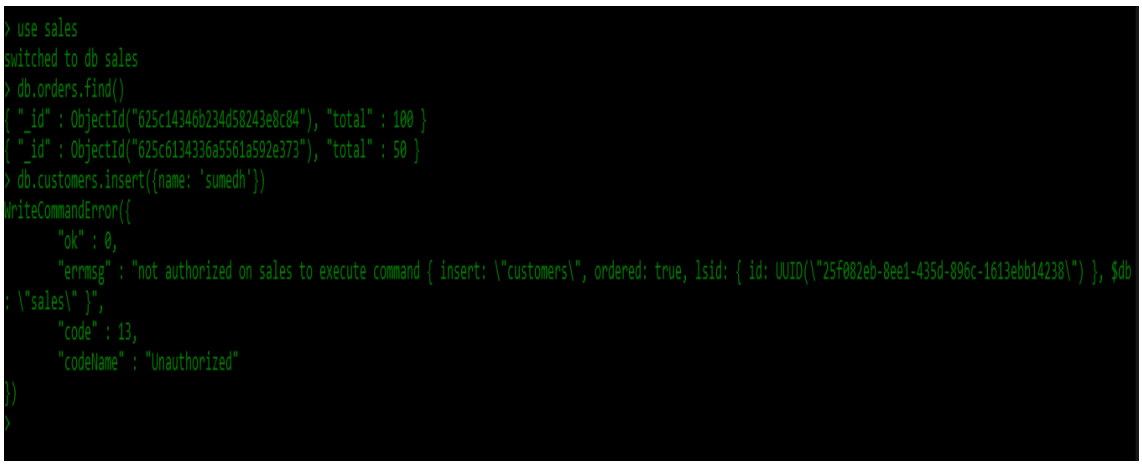

Fig. 9. Succesfull find and Failed insert in sales db

5 Conclusion

We explored a way for implementing ABAC in NoSQL databases in this paper, as well as a MongoDB implementation of the method. The implementation is modular, and it may be readily extended to any NoSQL database with RBAC support.

While the approach for deploying ABAC policies using RBAC roles covers a wide range of ABAC functionalities, it nevertheless falls short in a few areas. This incorporates all of ABAC's features..

It would be necessary to intercept each message delivered to the MongoDB server and have a proxy server connect to the server on behalf of the client in order to have a native implementation of ABAC in MongoDB. The MongoDB Wire Protocol governs the interaction between the client and the server in MongoDB. The MongoDB drivers transform the requests and responses to MongoDB Wire Messages. As a result of intercepting each communication, we can create a layer of access control in the proxy server, allowing native ABAC to be implemented. We intend to work on this in the future as well.

References

1. Sandhu R., Coyne E.J., Feinstein H.L., Youman C.E.:Role Based Access Control Models. IEEE Computer, vol 29, no 2, pp. 38–47, 1996.
2. Okman, L., Gal-Oz, N., Gonen, Y., Gudes, E., & Abramov, J. (2011). Security Issues in NoSQL Databases. 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, 541-547.
3. Colombo, P., & Ferrari, E. (2017). Towards a Unifying Attribute Based Access Control Approach for NoSQL Datastores. 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 709-720.
4. Li N., Tripunitara M.V.:Security Analysis in Role-Based Access Control. ACM Transactions on Information and System Security, vol. 9, no. 4, pp. 391—420, 2006.
5. Mohammad A. Al-Kahtani and Ravi S. Sandhu. A model for attribute-based user role assignment. In ACSAC, 2002.
6. Ali E. Abdallah and Etienne J. Khayat. A formal model for parameterized role based access control. In Formal Aspects in Security and Trust, 2004.
7. Mark Evered. Supporting parameterised roles with object-based access control. In HICSS, 2003.
8. Mei Ge and Sylvia L. Osborn. A design for parameterized roles. In DBSec, 2004.
9. Jeffrey Fischer, Daniel Marino, Rupak Majumdar, and Todd D. Millstein. Fine grained access control with object-sensitive roles. In ECOOP, 2009.
10. Jianming Yong, Elisa Bertino, Mark Toleman, and Dave Roberts. Extended RBAC with role attributes. In 10th Paci
c Asia Conf. on Info. Sys., 2006.
11. Jaehong Park and Ravi Sandhu. The UCONabc usage control model. ACM Trans. Inf. Syst. Secur., 2004.
12. Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In In 2nd ACM Workshop on FMSE, 2004.
13. Eric Yuan and Jin Tong. Attributed based access control (ABAC) for web services. In Intl. ICWS, 2005.