On automated RBAC assessment by constructing a centralized perspective for microservice mesh

Dipta Das[1], Andrew Walker[1], Vincent Bushong[1], Jan Svacina[1], Tomas Cerny[1] and Vashek Matyas[2]

[1] Department of Computer Science, Baylor University, Waco, TX, USA
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic

ABSTRACT

It is important in software development to enforce proper restrictions on protected services and resources. Typically software services can be accessed through REST API endpoints where restrictions can be applied using the Role-Based Access Control (RBAC) model. However, RBAC policies can be inconsistent across services, and they require proper assessment. Currently, developers use penetration testing, which is a costly and cumbersome process for a large number of APIs. In addition, modern applications are split into individual microservices and lack a unified view in order to carry out automated RBAC assessment. Often, the process of constructing a centralized perspective of an application is done using Systematic Architecture Reconstruction (SAR). This article presents a novel approach to automated SAR to construct a centralized perspective for a microservice mesh based on their REST communication pattern. We utilize the generated views from SAR to propose an automated way to find RBAC inconsistencies.

Subjects Security and Privacy, Software Engineering
Keywords Microservices, REST, RBAC, Access control, Authorization, Security, Static code analysis, Systematic architecture reconstruction

Submitted 18 August 2020
Accepted 6 January 2021
Published 1 February 2021

Corresponding authors
Tomas Cerny, tomas_cerny@baylor.edu
Dipta Das, dipta_das1@baylor.edu

Academic editor
Mario Luca Bernardi

Additional Information and Declarations can be found on page 20

DOI **10.7717/peerj-cs.376**

INTRODUCTION

With the software industry's growth, the complexity of security administration is becoming more and more challenging. As the current software development trend is moving rapidly from monolithic to MicroService Architecture (MSA), we must address communication patterns not only for the simple client to server scenarios but also for service to service scenarios. Since the client-server communication pattern has existed for many years, its security implications have already been well addressed. In contrast, not much has been studied for service-to-service communication patterns.

Currently, the most popular way to establish communication between services is to use Representational State Transfer (REST) (*Vural, Koyuncu & Guney, 2017*; *Salah et al., 2016*). Developing a secured REST-based infrastructure is relatively easy for a smaller number of microservices. However, the security aspects gradually become more complex as the number of microservices grows. Due to their high feature set and operational complexity, modern microservice-based applications tend to have a large number of individual microservices developed separately by separate teams. Enforcing a robust security solution for such applications is tedious for developers and might lead to security disagreement among microservices. This is because individual developers only have an idea of a subset of microservices they maintain but lack an understanding of the overall picture. Even system architects may not understand the complete picture of the application since many of those microservices may not be in the initial blueprint of the application but rather were added later.

Thus, we need to establish an automatic way to generate the overall communication pattern for the whole application before diving into the security aspects. This is done through a process of Systematic Architecture Reconstruction (SAR) in which overall views are constructed from existing application artifacts. SAR is divided into four phases: extraction, construction, manipulation and analysis.

In this article, we first introduce a solution for automatic SAR of a microservice application, which generates a view of the microservices' REST communication pattern. By automating the first three phases of SAR and utilizing the constructed views, we can focus on the analysis phase and present an approach to enumerate possible security loopholes in the application. More specifically, we focused on finding Role-Based Access Control (RBAC) inconsistencies among microservices using static code analysis. We present a case study on a single enterprise application called Teacher Management System (TMS) consisting of four individual microservices. This application was developed separately but re-purposed here as a testbed for performing static code analysis. Our work focuses on intra-and inter-microservice inconsistencies highlighting all possible RBAC issues.

An application's core security requirement is to ensure that it can only be used by legitimate users (*Mohanty, Mohanty & Balakrishnan, 2016*). RBAC is one of the popular methods of securing REST services where each user of the application is assigned to a set of roles that grant access to different parts of the system. In microservice-based applications, there can be two different abstractions to enforce RBAC rules. First, centralized among all the microservices and, second, per microservice-based.

Thus, next in this article, we focus on the centralized approach. Finding inconsistencies among RBAC rules in a large system is a cumbersome and difficult task due to different levels of abstractions, poor coding practices, and coupling with third-party services. According to a survey conducted in 2014 by the International Data Group (*Mohanty, Mohanty & Balakrishnan, 2016*), about 63% of applications have not been tested for security vulnerabilities. This can be easily mitigated by enforcing standard security features during the regular software development process (*McGraw, 2004*). Ignoring such security vulnerabilities is expensive. Security breaches can cost companies billions and require significant time and effort to resolve. For example, the 2014 eBay hack, which was caused by improper access control restrictions, impacted over 145 million users (*Swinhoe, 2020*). Being able to list possible security vulnerabilities automatically can significantly reduce the likelihood of such incidents.

System administrators should wisely choose the approaches to test the security vulnerability of the system. The most accurate outcome from such a test can be obtained via rigorous penetration testing. However, such an approach needs the application to be fully deployed, and running penetration tests against a production deployment could lead to disruption for end users. Also, it is difficult to emulate all possible scenarios for penetration testing. In contrast, static code analysis can be a much easier alternative that does not require an application to be deployed and hence is more cost-effective. Although static code analysis is no panacea, when carefully implemented, it can detect many vulnerabilities. It is for these reasons we use static code analysis for our automated SAR process.

The article is organized as follows. Section two discusses the related work and state of the art. "Proposed Method" describes our proposed method in detail, and section four explores a case study. Finally, we conclude our article by summarizing our work outcomes, describing our future goals, and listing the references. Throughout the article, the terms "inconsistency", "violation" and "issue" are used interchangeably to indicate a potential flaw.

RELATED WORK

In this section, we present related work from the two different perspectives considered in this article. First, we assess the limitations of RBAC analysis in the context of enterprise systems. Next, we assess existing approaches for the SAR.

Role-based access control

In microservice-based applications, each microservice implements a subset of features. End-users or other microservices can access these features through an application programming interface (API). There are typically two main API development choices: REST and Simple Object Access Protocol (SOAP) (*Tihomirovs & Grabis, 2016*). While REST is an architecture for API development that works over standard HTTP protocol, SOAP is just a protocol. For many years, SOAP was a standard approach for web service interfaces. However, it has been dominated by REST in recent years. According to Stormpath, over 70% of public APIs are designed using REST (*Hunsaker, 2015*). The main advantage of REST compared to SOAP is its simplicity and ease of learning. REST is lightweight and hence better suited for a wide range of devices, including mobile devices (*Wagh & Thool, 2012*). Apart from that, REST uses JavaScript Object Notation format which is faster to parse compare to Extensible Markup Language used in SOAP (*Tihomirovs & Grabis, 2016*; *Castillo et al., 2011*).

Securing REST API endpoints is generally easy when existing HTTP security approaches are leveraged instead of implementing a new security model (*Sudhakar, 2011*). Securing REST endpoints involves both authentication and authorization (*Brachmann, Dittmann & Schubert, 2012*). Authentication is the process of verifying the credentials associated with a particular request. Different enterprise applications use different strategies to authenticate incoming requests, such as basic authentication, token-based authentication, hash-based digest authentication, OAuth, etc. (*Lee, Jo & Kim, 2015*). On the other hand, authorization involves verifying whether a request connection is allowed to perform a particular action through a REST endpoint. Mandatory access control, discretionary access control, attribute-based access control and RBAC are popular approaches for enforcing authorization (*Sandhu & Samarati, 1994*). In this article, instead of authentication breaches, we focus on exploring and detecting possible authorization inconsistencies, specifically role-based authorization inconsistencies.

Role-Based Access Control has been widely adopted as an alternative to classical discretionary and mandatory access controls because of its advancement in flexibility and detail of control (*Sandhu & Samarati, 1994*; *Sandhu et al., 1996*). It regulates users' access to information and system resources based on activities that users need to execute in the system and requires the identification of roles in the system (*Ahn & Sandhu, 2000*). RBAC's administrative capabilities have made it stand out from the alternative approaches because system administrators can statically or dynamically regulate user's access by defining roles, role hierarchies, relationships, and constraints (*Ferraiolo, Cugini & Kuhn, 1995*). For distributed systems, another advantage is that RBAC administrative responsibilities can be divided into central and local protection domains (*Ferraiolo, Cugini & Kuhn, 1995*). In the case of microservice-based applications, these can be translated into central policies for all associated microservices and per microservice-based policies. Central RBAC policies can be enforced by delegating authentication and authorization tasks to a separate identity management tool, such as Red Hat's Keycloak (*Red Hat Inc, 2020a*). On the other hand, individual microservices can carry out such policies using security features of underlying frameworks, such as spring-security for spring-based applications (*Scarioni & Nardone, 2019*).

Due to the high impact of security-related issues, much research and development have been done addressing role violations. *Ciuciu, Tang & Meersman (2012)* described one such strategy where appropriate security annotations are recommended for developers based on the ontology extracted from the business information. However, since this recommendation strategy works only based on business information irrespective of source code, if the business information provided is flawed, then the recommendation will also be faulty.

One similar study focused on finding security vulnerabilities of API implementations among different libraries based on security-sensitive events (*Srivastava et al., 2011*). It finds discrepancies among security policies associated with the same API using a flow graph. The inherent drawback of this approach is that it requires multiple independent implementations of the same API, and it can not find which ones of whose multiple implementations are faulty. Another research study described a model-based approach for testing access control rules based on consistency, completeness and redundancy (*Xu et al., 2012*). It checks whether access control rules are consistent across the methods, whether they are unnecessarily repeated, and whether they covered all subset of permissions. However, the coverage of access control rules over a set of methods does not necessarily relate to security issues. In *Xu et al. (2012)*, the system under study does not allow a user to rent a book on maintenance due to the incompleteness of access control rules, which is more of a system flaw rather than a security issue. In contrast, our proposed method finds whether a user can access a resource-restricted by one RBAC rule through an alternative path that has less restriction.

The tool FixMeUp (*Son, Mckinley & Shmatikov, 2013*) proposed an automated way to fix access control issues in PHP applications using static code analysis. It automatically edits the source code to resolve access control issues. Although it seems compelling to automate the task, it might lead to syntax errors and might result in unintended consequences in case of false positives. On the other hand, our RBAC tool pinpoints the location of possible inconsistencies in the source code without adversely affecting the codebase since it does not modify the source code while performing analysis.

The most similar analysis to our proposed method has been described by *Walker et al. (2020)*. That tool performs static code analysis on enterprise JAVA applications to find issues in RBAC rules defined using security annotations. The key difference is that it only considers intra-microservice issues, while our method works for both intra- and inter-microservice issues, taking into account all the microservices that constitute the application.

Freudenthal et al. (2002) proposed a distributed RBAC (dRBAC) mechanism that decentralizes the trust-management across multiple administrative domains. Due to its distributive nature dRBAC is highly scalable for a large number of mutually anonymous users. It features third-party delegation that enables one authorized entity to entrust roles created by another entity. Besides, it controls the access levels for the same resource by valued attributes. Also, dRBAC presents continuous monitoring by utilizing a pub-sub model to ensure the validity of trust relationships for extended interactions. In this article we do not assess such decentralized RBAC techniques, rather we assume that the user authentication and role mapping are handled through a centralized service while individual microservices are responsible for the imposition of those roles on API endpoints.

Separation of Duties (SoD) has been widely studied in the context of RBAC. It ensures data integrity and fraud prevention by distributing critical tasks among multiple users (*Basin, Burri & Karjoth, 2009*). It enforces that no single user can execute all actions and thus any kind of fraudulent activity will cause collision among at least two users (*Habib et al., 2014*). In RBAC, SoD can either static or dynamic (*Sandhu, 1990*). In the static separation of duties (SSD) constraints are enforced during the assignment of users to roles. On the other hand, in dynamic separation of duties (DSD) constraints are activated on the roles within a user session (*Omicini, Ricci & Viroli, 2005*). In this article, we are not considering the user assignments and user sessions. Instead, we performed static code analysis that solely focused on a subset of SSD including statically defined roles and role hierarchies.

Software architecture reconstruction

Although many studies address access control issues, most of them are focused on single microservice or monolith applications. However, modern cloud-based applications are commonly designed as a set of microservices for better flexibility and scalability (*Salah et al., 2016*). The key challenge to perform a holistic analysis across multiple microservices is the automated construction of the application's centralized perspective. SAR extracts a representation of software architecture from source code or documentation through an iterative reverse engineering process (*Bass, Clements & Kazman, 2003*).

It is historically defined with four phases: extraction, construction, manipulation and analysis (*Bass, Clements & Kazman, 2003*). In the extraction phase, all necessary artifacts are collected. Each set of related artifacts is relevant to a *view* that represents relations among certain elements of the software architecture (*Bass, Clements & Kazman, 2003*). The construction phase creates canonical representations of the views. Then the manipulation phase combines the views to create a more compact representation to answer specific questions in the analysis phase. Lastly, the analysis phase answers specific research questions from the reconstructed views. In this article, the analysis phase addresses the detection of possible RBAC inconsistencies. Also, to the best of our knowledge SAR has not been used to detect RBAC inconsistencies in MSA.

One approach of SAR of microservice-based systems is described by *Rademacher, Sachweh & Zündorf (2020)*. This method describes different modeling based on different viewpoints (*Rademacher et al., 2020*) where domain modeling is based on bounded context, services modeling is based on REST calls, and operation modeling is based on deployment specifications, for example, Dockerfiles.

The Model-Driven Engineering (MDE) approach is commonly used in SAR. In MDE, models are used as first-class entities to depict an efficient representation of the software architecture (*Cicchetti et al., 2013*). *Alshuqayran, Ali & Evans (2018)* described a manual analysis through the MDE approach to reconstruct the architecture of microservice-based open-source projects. They defined a metamodel which is then mapped to the architecture using mapping rules. The metamodel and mapping rules are initially created for one system and then refined and validated using seven additional systems. However, the authors did not apply their reconstruction strategy to answer specific questions.

Ibrahim, Bozhinoski & Pretschner (2019) proposes an approach to derive MSA module topology from container-based deployment configuration files, more specifically, from Docker Compose files. In addition to topology, they extracted the *attack graph*, a directed acyclic graph, to identify attack paths that lead to vulnerability exploitation. Their implementation is based on a open-source vulnerability scanner for Docker containers named Clair (*Quay, 2020*).

The MicroART tool described by *Granchelli et al. (2017)* extracts the deployment architecture of a microservice-based system from the source code repository. It utilizes a domain-specific language to represent key elements of the architecture by using the MDE approach. It employs runtime log analysis to discover containers, network interfaces, and service interactions. However, users need to provide a reference to the container engine since MicroART does not automatically detect it from deployment configuration files.

Our proposed method reconstructs MSA architecture based on the REST communication pattern, similar to the service modeling described by *Rademacher, Sachweh & Zündorf (2020)*. However, unlike that system, which depends on a Service Modeling Language (*Rademacher et al., 2020*), our reconstruction is solely based on static code analysis and works independently.

PROPOSED METHOD

Enterprise applications are typically organized into a three-layer structure: controller layer, service layer, and repository layer. It is also common to organize microservices into the presentation layer, business layer, persistence layer, and database layer (*Richards, 2015*). These two commonly used structures essentially indicate the same strategy. The presentation layer maps to the controller layer, which defines API endpoints, and the business layer maps to the service layer, which contains business logic. The persistence layer maintains data access objects to interact with the database layer (*Alur et al., 2003*). These two layers (persistence and database) are consolidated into the repository layer in the three-layer architecture (*Richards, 2015*; *Steinegger et al., 2017*).

Microservices typically communicate over REST APIs (*Salah et al., 2016*). Each microservice's controller layer defines the REST endpoints that serve as request entry points for that particular microservice. Requests are delegated from the controller layer to the service layer. The service layer typically implements business logic. It processes the request and generates an appropriate response. The service layer can also incorporate with the persistence layer to store and retrieve data relevant to a specific request. However, sometimes the service layer depends on other microservices to process the request. In that case, it creates REST calls to other microservices and implements business logic based on the response. This describes a typical REST communication scenario among microservices. In particular, the service layer of one microservice makes REST calls to other microservice's controller layers to implement its business logic. Thus, the REST endpoints of each microservice can be either accessed by end-users or other microservices.

Enterprise frameworks adopted annotation-based configuration to define REST endpoints, for example, `@RestController` annotation in Spring-based JAVA applications and `@app.route` annotation in Flask based Python applications (*VMware Inc, 2020*; *Pallets Projects, 2020*). Since the REST endpoints are the entry points to the microservice, securing them is the single most important task for the developers. While there are several ways to enforce role-based authorization, the most widely adopted method in enterprise applications is to define authorization realms through the application server (*Oberle et al., 2004*) or through separate identity management tools like Keycloak (*Red Hat Inc, 2020b*). A realm is a security policy domain defined in the application server that contains a collection of users (*Jendrock et al., 2014*). These users might be further organized into several groups (*Jendrock et al., 2014*). Centralized authorization systems like Keycloak handles user authentication and role mapping. But such systems do not verify whether developers properly enforced RBAC policies during API implementation or not. For example, some API endpoints might have missing RBAC roles. In that case, any authenticated user can access those endpoints. Similarly, two API endpoints with different roles might eventually access the same entity which might be unintentional and left unnoticed. These inconsistencies are not flagged by the centralized authorization system and thus defining authorization policies are not enough to secure the endpoints. Developers need to enforce those policies within the application's source code that runs in that application server. Designing proper authorization policies are just one part of ensuring robust RBAC enforcement, we need to consider coding problems that might lead to security loopholes. In this article, we focused on detecting such coding problems through static code analysis. Also, it is important to classify these problems to understand the severity and origin of them. We have defined five types of possible inconsistencies or violations:

1. Missing role violations: this type of violation occurs when an API endpoint does not have any roles associated with it. In this case, all authenticated users can access the endpoint. Such violation typically happens when developers forget to enforce authorization roles on an API endpoint. However, it could be false-positive, for example, some API endpoint might be intentionally left open for all users.
2. Unknown access violations: if an API endpoint contains an authorization role that is not present in the user-defined role hierarchy, then we define it as an unknown access violation. Usually this type of violation results from typographical errors and in most cases, such typos are left unnoticed since they do not cause any compilation errors. As a result, legitimate users with proper access are denied from accessing the endpoint.
3. Entity access violations: if input and output that is, request and response types of two API endpoints are similar but they have different authorization roles, then we classify it as an entity access violation. This kind of violation indicates that the same entity is being accessed by users with different access roles.
4. Conflicting hierarchy violations: this type of violation happens when an intermediate method in the service layer or repository layer contains two different roles that are ancestor of each other's in the role hierarchy. This violation signifies that users with a junior role are accessing some functionalities that might be intended for users with a senior role (*Walker et al., 2020*).
5. Unrelated access violations: similar to conflicting hierarchy these violations focus on intermediate methods instead of endpoint methods. When an intermediate method contains two multiple roles that are located in different subtrees of the role hierarchy, we classify it as an unrelated access violation. This type of violation indicates poorly separated concerns while distributing access roles across different functionalities of the application (*Walker et al., 2020*).

Like REST configurations, authorization policies are typically applied by annotating methods or functions with appropriate security annotations. These annotations can differ based on the framework used to develop the application; for example, JAX-RS security annotations are used with JAVA EE based application (*Oracle, 2020*). A similar approach to enforce RBAC using annotation can also be found in Python applications based on the Flask framework (*Thio, 2020*). These security annotations define the level of restrictions applied to the associated methods or functions. Table 1 highlights the most commonly used security annotations in JAVA EE applications supported by JSR 250 (*Mordani, 2016*; *Oracle, 2020*).

For example, if we add a `@rolesAllowed(ADMIN)` annotation to a controller endpoint method, only the users that have the "ADMIN" role (defined in the realms) can access the

Table 1 **JAVA EE security annotations.**

Annotation	Description
`@permitAll`	All security roles are permitted
`@denyAll`	No security roles are permitted
`@rolesAllowed`	List of permitted security roles

endpoint. However, since the number of such endpoint methods can be significant and can grow over each iteration of the development cycle, it is possible to introduce inconsistencies among the allowed roles or even missing roles. Moreover, since these inconsistencies and missing roles do not cause any compilation or run time error, it is tempting for developers to overlook them, and that might result in potential security loopholes.

Our proposed method analyzes a set of microservice artifacts that communicate with each other through REST calls. It finds potential RBAC violations for the whole microservice mesh by scrapping security metadata of individual microservices and by combining them based on their REST communication flow. We divided the analyzer into three modules: a discovery module, a flow-matcher module, and an analysis module. The discovery module implements the extraction phase of SAR. It collects endpoint specification and security metadata. Next, the flow-matcher module performs the construction and manipulation phases of SAR by resolving the interaction among microservices. Finally, the analysis module completes the analysis phase of SAR and detects potential RBAC violations based on the other two modules' output.

The discovery module performs static code analysis on individual microservice artifacts. It detects the REST endpoints and security roles attached to those endpoints. Apart from that, it also lists the REST calls to other microservices, which are typically implemented in the service layer. The discovery module works for both source code artifacts and bytecode artifacts (e.g., JAR file, Python bytecode) and thus provides generalization for both interpreted languages (e.g., JAVA, Python) and compiled languages (e.g., C++, Go). The source code version of the discovery module takes a microservice artifact as input and parse class definitions while the bytecode version does the same using bytecode analysis. As discussed above, both REST endpoints and security policies are typically defined using the annotation-based configuration in enterprise applications. The descriptions of these annotations are well structured and preserved in the source code and in the bytecode. The discovery module scans each class to find REST annotations and security annotations that define REST endpoints and security roles, respectively. It aggregates class-level annotations with method-level annotations to derive the complete definition of each REST endpoints. It collects the allowed roles, port, path, HTTP type, type of request object, and type of response object for each endpoint. It takes account of all standard HTTP types, with the most commonly used ones being *GET, POST, PUT* and *DELETE*. The discovery module then further analyzes service layer classes to detect REST calls to other microservices. For each REST call, it detects the URL, HTTP type, type of request object, and type of response object. It parses REST client definitions to gather those attributes.

However, detecting the URL string involves further intensive analysis since the URL string is usually constructed by performing consecutive append operations in different parts of the source code. For this, our discovery module applies a backward recursive data flow analysis from the point where the URL is used to the point where the URL was initialized. In each intermediate step of the data flow where the URL was referenced, it scans for any append operations and resolves them to restore the final URL. Parts of the URL may also be constructed using values defined in the configuration files instead of hardcoded strings within the source code. Our module also scans configuration files of the project to resolve those values. Finally, the discovery module generates method-call graphs for individual microservices. It takes each controller method as the root node and populates child nodes by traversing subsequent method calls to the service layer and repository layer methods. For each microservice, the discovery module organizes all the scrapped information described above into a usable structure and passes them to the flow matcher module and analysis module.

As discussed by *Walker et al. (2020)*, RBAC security analysis for individual microservices is insufficient. It fails to acknowledge violations when an end-user gains access to a normally restricted resource by creating a proxy request through another microservice mediating the resource access through service layer REST calls. To detect such violations, we need to consider the whole MSA mesh instead of a single MSA, and we need to resolve REST communications between them to construct a complete centralized perspective. In our proposed model, the flow matcher module constructs the centralized communication graph for the whole MSA mesh. It takes descriptions of REST-endpoints and REST-calls for each microservice prepared by the discovery module. It combines all the REST endpoints into a list and all the REST calls into another list. Then it performs a brute force matching between those two lists to resolve all REST communications among the microservices. This involves matching the URL (including port and path), HTTP method, request type, and response type.

However, it is common for modern microservices to use service discovery and service registry instead of a hardcoded IP address in the URL (*Montesi & Weber, 2016*). To resolve this, our flow matcher module matches both the IP address and service name and checks if one of them matches. The service name is usually defined in each microservices' configuration files and scrapped during the discovery phase. The flow matcher module also generates a diagram of REST communication for the whole microservice mesh for better visualization.

The analysis module takes descriptions of method-call graphs and allowed roles from the discovery module and REST communication descriptions from the flow matcher module. Additionally, it takes the role hierarchy tree from the user. Figure 1 shows a user-defined role tree passed to the analysis module as input. Roles higher in the hierarchy tree are senior to the roles below in the tree; senior roles should have all the access rights junior roles have, plus additional rights the junior roles do not have. Roles in separate paths of the hierarchy are not related to each other. The analysis module

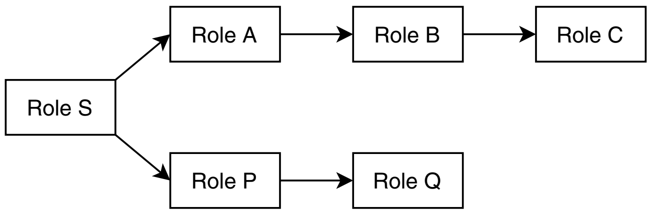

Figure 1 A sample user-defined role hierarchy tree. Senior roles are higher in the tree; in this example, Role S is the most senior role, Role A is senior to B, which is senior to C. Role P is senior to Q.
Full-size DOI: 10.7717/peerj-cs.376/fig-1

combines method-call graphs of different microservices based on their REST communication. Figure 2 depicts a typical scenario of how combined method-call graphs are constructed. Each node of the combined graphs can be a controller node, service node, or repository node. Typically, only the controller nodes contain RBAC information, that is, a list of allowed roles; however, the service layer and repository layer nodes can also have RBAC information. To find potential RBAC violations in those layers, the analysis module loops through all the nodes and analyzes the roles associated with them. The first three types of violations are only related to controller nodes. If any controller node does not have any roles associated with it, we detect it as a missing role violation. This is the most common type of violation that might occur since missing roles on controller methods do not cause any compilation errors. If a node contains a role that is not defined in the user-provided role hierarchy, we detect it as an unknown access violation. This type of violations typically results from typographical errors. If request types, response types, and HTTP types of two controller methods are equal, but they have different RBAC roles, we detect it as an entity access violation. This violation implies similar access to a particular entity with different roles.

The unrelated access and conflicting hierarchy violations occur when a node contains multiple roles after performing the reduction and aggregation. In the reduction phase, the analysis module goes through each node and keeps the lowest role defined in the user-provided role hierarchy. The significance of this reduction is that it defines the minimum role required to access a specific part of the application. After reduction, in the aggregation phase, the analyzer traverses each graph and copies the allowed role from the parent node to the child node. If a child node contains an RBAC role or a child node has multiple parents with different roles, then it aggregates the roles for that particular child node. Figure 3 shows how the analysis module labels each child node using the RBAC roles of its parent nodes according to the role hierarchy shown in Fig. 1.

The conflicting hierarchy violation occurs when a node code contains two different roles where one role is an ancestor of another role in the user-defined role hierarchy. This violation indicates a place where a junior role potentially accesses an area reserved for a more senior role. It is only a potential violation because it is ambiguous whether a junior role is accessing an area reserved for a senior role or the senior role is accessing something allowed for the junior role (*Walker et al., 2020*). The unrelated access violation is the opposite of the conflicting role violation. It happens when a node contains two roles

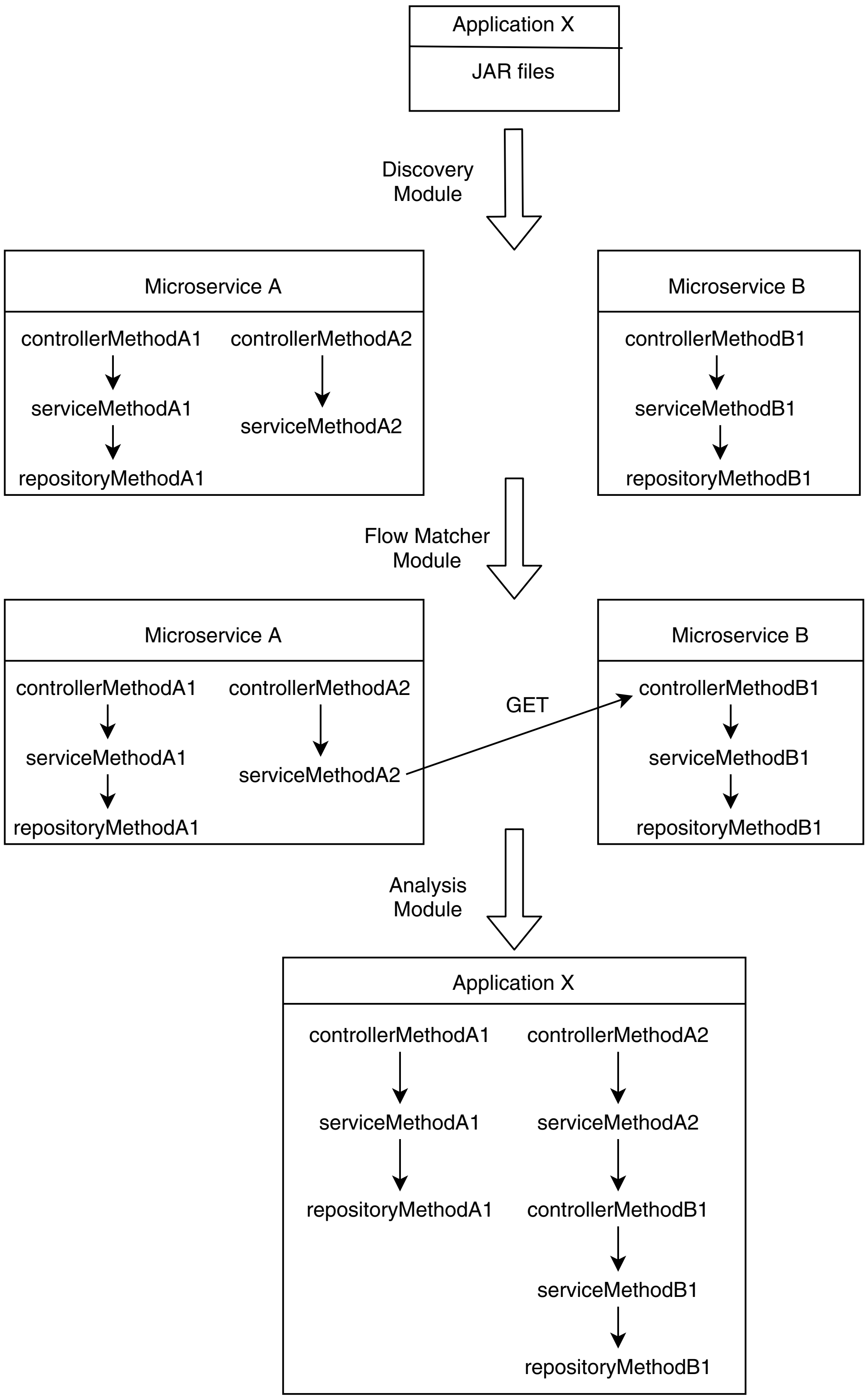

Figure 2 **Construction of combined method-call graphs.**

Full-size DOI: 10.7717/peerj-cs.376/fig-2

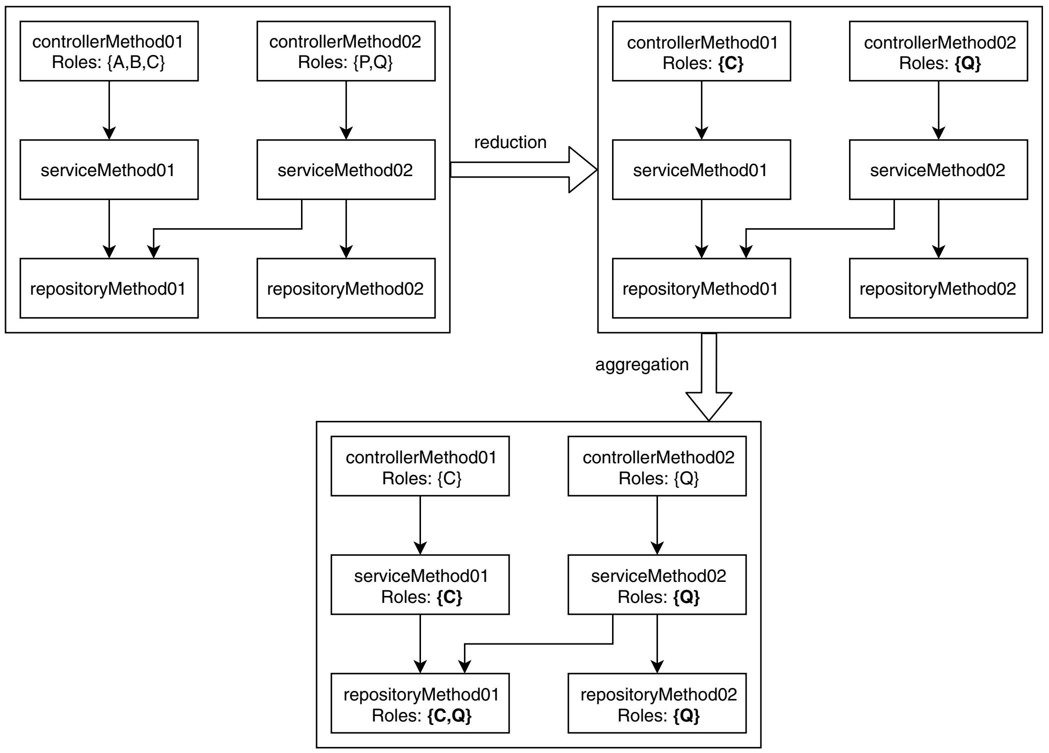
controllerMethod01 Roles: {A,B,C}
controllerMethod02 Roles: {P,Q}
serviceMethod01
serviceMethod02
repositoryMethod01
repositoryMethod02
reduction
controllerMethod01 Roles: {C}
controllerMethod02 Roles: {Q}
serviceMethod01
serviceMethod02
repositoryMethod01
repositoryMethod02
aggregation
controllerMethod01 Roles: {C}
controllerMethod02 Roles: {Q}
serviceMethod01 Roles: {C}
serviceMethod02 Roles: {Q}
repositoryMethod01 Roles: {C,Q}
repositoryMethod02 Roles: {Q}

Figure 3 Reduction and aggregation of RBAC roles. Full-size DOI: 10.7717/peerj-cs.376/fig-3

located in a different subtree of the user-defined role tree, that is, one role is not an ancestor of another role. This violation indicates areas where unrelated roles are accessing the same application area, which may indicate poorly separated concerns that could be refactored (*Walker et al., 2020*). For example, considering the role hierarchy shown in Fig. 1, if a node has roles {A, C} then it is detected as a conflicting hierarchy violation, and if a node has roles {A, P} then it is marked as an unrelated access violation. The categorization of violations defined in our proposed method is mostly similar to the ones discussed by *Walker et al. (2020)*. However, *Walker et al. (2020)* only considered only a single microservice at a time, whereas we also analyze inconsistencies across microservices.

Our system finds potential RBAC violations based on a user-defined role hierarchy for the whole microservice mesh (a set of microservices). It warns the developer about potential violations by providing a report of specific locations where the violations are detected and the categories, as discussed above. While some of the detected violations may be false-positive and intentional, our proposed method provides an overall idea of all possible RBAC violations for a large and complex system. The categorization of the violations helps the developer understand each violation's severity, while the specific locations of the violations help to find and fix them easily.

CASE STUDY

The TMS[1] is an enterprise application developed at Baylor University for Central Texas Computational Thinking, Coding, and Tinkering to facilitate the Texas Educator

[1] https://github.com/cloudhubs/tms2020.

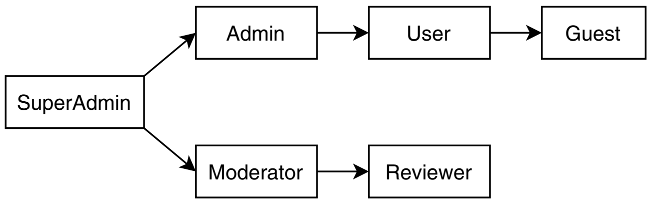

Figure 4 **Role hierarchy tree of the TMS application.** Full-size DOI: 10.7717/peerj-cs.376/fig-4

Table 2 **Annotations used in TMS project.**

Annotation	Target	Description
`@Controller`	3 Class	3 Indicates controller, service, and repository layers
`@Service`		
`@Repository`		
`@RestController`	Class	Sub type of `@Controller` to activate REST APIs
`@RequestMapping`	Class and Method	Defines HTTP types and paths for REST endpoints
`@GetMapping`	3 Method	3 Sub types of `@RequestMapping` for specific HTTP types
`@PostMapping`		
`@DeleteMapping`		
`@RolesAllowed`	Method	Lists a set of allowed roles

Certification training program. The whole TMS system consists of four individual microservices: user management system (UMS), question management system (QMS), exam management system (EMS) and configuration management system (CMS). All of the microservices are developed using the Spring Boot framework (*Walls, 2016*) and structured into the controller, service, and repository layers. The RBAC authorization is enforced using annotations on each controller method for the individual microservices, while the central authentication and authorization policies are defined using Keycloak (*Red Hat Inc, 2020a*). Figure 4 shows the role hierarchy tree for the TMS application. For our case study, we added mutants (*Jia & Harman, 2011*) for each type of violations that resulted in a total of seven RBAC violations. Our system successfully detected all those violations and provided a report with specific locations of the violations. In this section, we will discuss how our analysis process works in detail for the mutated application.

The TMS project utilizes an annotation-based configuration technique to define application layers. REST API configurations and RBAC restrictions are also applied through annotations, which are common practice for enterprise applications. Table 2 lists frequently used annotations throughout the TMS project.

For our purpose, we only looked for the `@RestController` annotation in the discovery module. The HTTP and paths type were extracted from the parameters of `@RequestMapping` annotation or subtype annotations. Paths can be defined at both class level or method level. We aggregated the class level paths with method-level paths to get the final path for each endpoint. The endpoints' request and response types are

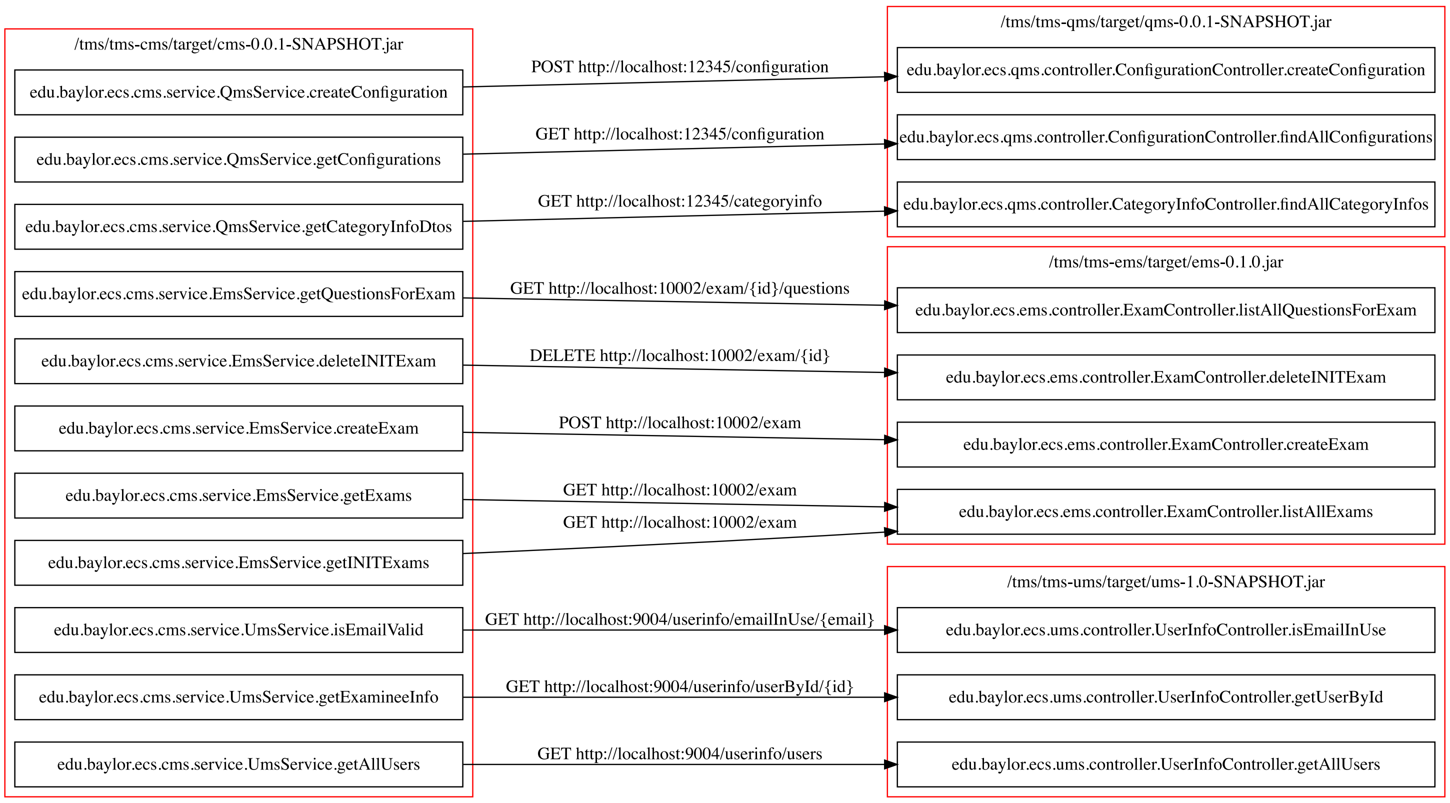
/tms/tms-cms/target/cms-0.0.1-SNAPSHOT.jar
edu.baylor.ecs.cms.service.QmsService.createConfiguration
edu.baylor.ecs.cms.service.QmsService.getConfigurations
edu.baylor.ecs.cms.service.QmsService.getCategoryInfoDtos
edu.baylor.ecs.cms.service.EmsService.getQuestionsForExam
edu.baylor.ecs.cms.service.EmsService.deleteINITExam
edu.baylor.ecs.cms.service.EmsService.createExam
edu.baylor.ecs.cms.service.EmsService.getExams
edu.baylor.ecs.cms.service.EmsService.getINITExams
edu.baylor.ecs.cms.service.UmsService.isEmailValid
edu.baylor.ecs.cms.service.UmsService.getExamineeInfo
edu.baylor.ecs.cms.service.UmsService.getAllUsers
POST http://localhost:12345/configuration
GET http://localhost:12345/configuration
GET http://localhost:12345/categoryinfo
GET http://localhost:10002/exam/{id}/questions
DELETE http://localhost:10002/exam/{id}
POST http://localhost:10002/exam
GET http://localhost:10002/exam
GET http://localhost:10002/exam
GET http://localhost:9004/userinfo/emailInUse/{email}
GET http://localhost:9004/userinfo/userById/{id}
GET http://localhost:9004/userinfo/users
/tms/tms-qms/target/qms-0.0.1-SNAPSHOT.jar
edu.baylor.ecs.qms.controller.ConfigurationController.createConfiguration
edu.baylor.ecs.qms.controller.ConfigurationController.findAllConfigurations
edu.baylor.ecs.qms.controller.CategoryInfoController.findAllCategoryInfos
/tms/tms-ems/target/ems-0.1.0.jar
edu.baylor.ecs.ems.controller.ExamController.listAllQuestionsForExam
edu.baylor.ecs.ems.controller.ExamController.deleteINITExam
edu.baylor.ecs.ems.controller.ExamController.createExam
edu.baylor.ecs.ems.controller.ExamController.listAllExams
/tms/tms-ums/target/ums-1.0-SNAPSHOT.jar
edu.baylor.ecs.ums.controller.UserInfoController.isEmailInUse
edu.baylor.ecs.ums.controller.UserInfoController.getUserById
edu.baylor.ecs.ums.controller.UserInfoController.getAllUsers

Figure 5 **Inter microservice REST communications in TMS.** Full-size DOI: 10.7717/peerj-cs.376/fig-5

resolved by detecting parameters and return types of respective methods where the endpoints are defined. Finally, the RBAC roles are listed by detecting the parameters of the `@RolesAllowed` annotation applied to each endpoint method.

The `RestTemplate` class is usually used for making REST calls in the Spring Boot applications where the methods `getForObject`, `postForObject`, `deleteForObject`, etc. are used for performing REST calls with specific HTTP type. Each of those methods takes a URL parameter and a request object and returns a response object. We scan classes annotated with `@Service` annotation and filter them if they contain `RestTemplate` in their import statements to detect service layer REST calls. We then look for the methods described above and detect request and response types by checking the parameter type and return type. The URLs are detected by performing a backward data flow analysis recursively, as described in the proposed method section. The method calls graph is constructed by traversing each endpoint method to the service layer and repository layer methods.

After the discovery module completes gathering metadata for each MSA, the flow matcher module combines them, and the analysis module performs the final analysis. The flow matcher module also generates a visual graph of the REST communications among the microservices using Graphviz library (*Ellson et al., 2002*). Figure 5 shows the generated graph for the TMS application.

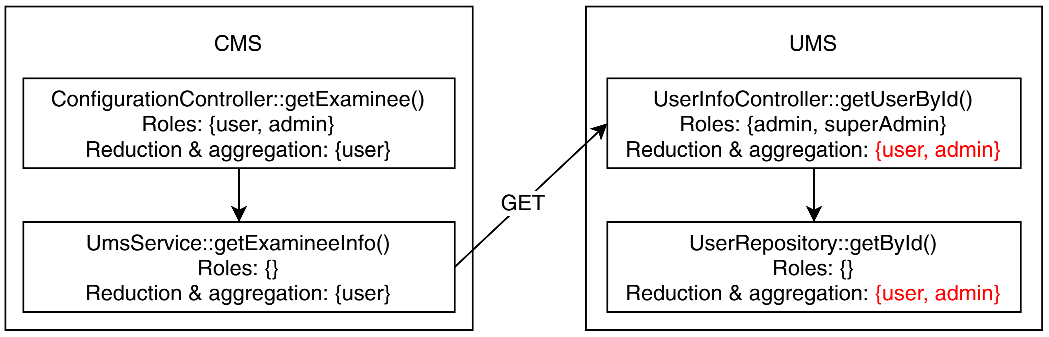
CMS
ConfigurationController::getExaminee()
Roles: {user, admin}
Reduction & aggregation: {user}
UmsService::getExamineeInfo()
Roles: {}
Reduction & aggregation: {user}
GET
UMS
UserInfoController::getUserById()
Roles: {admin, superAdmin}
Reduction & aggregation: {user, admin}
UserRepository::getById()
Roles: {}
Reduction & aggregation: {user, admin}

Figure 6 Conflicting hierarchy violation among CMS and UMS.
Full-size DOI: 10.7717/peerj-cs.376/fig-6

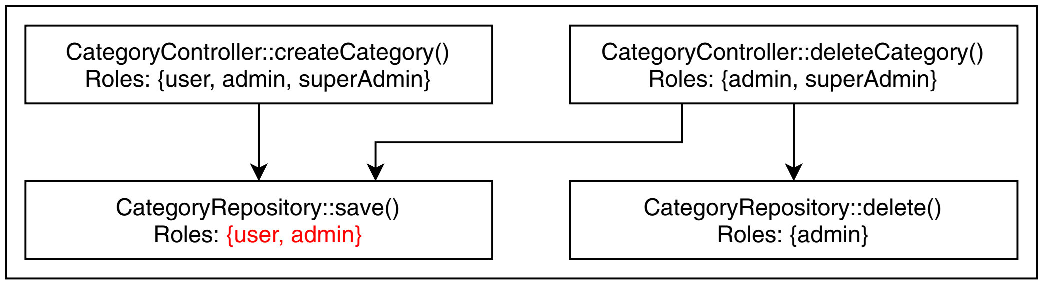
CategoryController::createCategory()
Roles: {user, admin, superAdmin}
CategoryController::deleteCategory()
Roles: {admin, superAdmin}
CategoryRepository::save()
Roles: {user, admin}
CategoryRepository::delete()
Roles: {admin}

Figure 7 Conflicting hierarchy violation within QMS. Full-size DOI: 10.7717/peerj-cs.376/fig-7

While matching the request and response types, we only considered the supertype of the generic types. For example, `List<AClass>` and `ArrayList<AClass>` are considered equal during matching.

Our analyzer reported two missing-role violations for the mutated applications by specifying the fully qualified name (MSA name + package name + class name + method name) of the endpoint methods that are defined without specifying any RBAC roles. It detected two unknown-role violations along with their locations. These two violations have resulted from data entry errors where “user” and “admin” roles are mistakenly typed as “usre” and “admin” respectively, which are not present in the role hierarchy shown in Fig. 4. Our analyzer flagged one entity access violation by pointing out a pair of fully qualified method names. Methods `getExams` and `getINITExams` in CMS have the same return type `List<Exam>` and the same HTTP type `GET` but they have different RBAC roles: “user” and “moderator” respectively.

We found two conflicting hierarchy violations in the mutated TMS application. One of them occurred in inter microservice communication, shown in Fig. 6, where the CMS module calls the UMS module to retrieve examinee info. The `getExaminee` endpoint method in CMS can be accessed with a “user” role which calls the `getUserById` endpoint method of EMS via service layer REST call. However, the `getUserById` method in EMS has annotated with the “admin” role, which is a direct ancestor of the “user” role. The second conflicting hierarchy violation, shown in Fig. 7, occurred entirely within the QMS module where both `createCategory` and `deleteCategory` endpoint methods

Table 3 **Runtime against TMS testbed.**

Module	Total	Time breakdown	
Name	**Runtime (s)**	**MSA**	**Time (s)**
Discovery	1.04	CMS	0.43
		EMS	0.18
		QMS	0.31
		UMS	0.12
Flow Matcher	0.13	–	
Analysis	0.29	–	

call the `save` method of `CategoryRepository` with conflicting roles. Finally, we detected one unrelated access violation between CMS and EMS, where the method `getQuestions` in CMS has transitive access to the method `listAllQuestionsForExam` in EMS via service layer REST call. They are annotated with "user" and "moderator" roles, respectively defined in separate subtrees of the role hierarchy.

We tested both source code and bytecode version of our discovery module, which utilizes the JavaParser library (*Bruggen, 2020*) to parse the source code and JavaAssist library (*JBoss, 2020*) to perform bytecode analysis to extract class definitions. We published our implementation as an open-source tool[2,3,4]. We ran it against the TMS project for benchmarking our analyzer and separately measured the runtime for each discovery, flow matcher, and analysis modules. For the discovery module, we break down our measurements for each microservice (CMS, QMS, EMS and UMS) and count the number of classes it scanned. Note that the discovery module performs a deep scanning for the controller layer classes that are annotated with `@RestController` annotation and service layer classes that have `RestTemplate` import to detect REST endpoints, security roles, and REST calls. For other classes, it performs just a shallow scan to construct the method call graphs.

Table 3 shows the total runtime[5] for each module and the breakdown for the discovery module for static bytecode analysis. We can immediately see that the discovery module takes the most significant time since it performs scanning of all class files to extract metadata. In contrast, the flow-matcher and the analysis module, operating on the extracted metadata, take comparatively less time. For the discovery module, runtime depends on the number of class files in each microservices. The runtime of the flow-matcher module depends on the number of REST endpoints and the number of REST calls, while the runtime of the analysis module depends on the number of inter-microservice REST connections and the depth of the function call graph.

Our experiment exhibits a reasonable runtime to perform the static code analysis for enterprise applications. In total, it took 1.43 seconds against the TMS application, which consists of four microservices, a total of 102 classes, and 11 inter-microservice REST connections. For enterprise applications with many microservices, it is possible to run the discovery module in parallel for multiple microservices, which will significantly reduce the runtime.

[2] SAR from bytecode: https://github.com/cloudhubs/rad.

[3] SAR from source code: https://github.com/cloudhubs/rad-source.

[4] RBAC analysis: https://github.com/cloudhubs/rad-analysis.

[5] The benchmark is run on a Mac OS computer with a 2.9 GHz 6-core Intel Core i9 processor and 32 GB RAM.

```
RBACAssessment(pathToMicroservices, roleHierarchy) {
    // extract metadata
    for each path in pathToMicroservices {
        analyze project property files to get service-name,
            port, hard-coded string values, etc.

        extract class definition using static code analysis

        // populate serverList and clientList
        for each class {
            for each method {
                if the method annotated with REST annotations {
                    extract API endpoint definition metadata

                    add extracted metadata to serverList

                    follow each method call to create a method call graph

                    extract RBAC security roles associated with those methods

                    add the graph to methodCallGraph as a subgraph
                }
                if the method contains REST API calls {
                    extract API call descriptions e.g. HTTP method, URL, etc.

                    add extracted metadata to clientList
                }
            }
        }
    }
    // resolve inter-microservice REST connections
    for each server in serverList {
        for each client in clientList {
            if URL, port, HTTP method matches for server and client {
                add (server, client) pair to restConnections
            }
        }
    }
    // update method call graph
    for each connection in restConnections {
        add an edge from client to server in methodCallGraph
    }
    // reduction
    for each method in methodCallGraph {
        keep only the lowest role in roleHierarchy and discard others
    }
    // aggregation
    for each disjoint subgraph in methodCallGraph {
        traverse all paths and merge the roles from parent to child
    }
    // find inconsistencies
    for each method in methodCallGraph {
        if the method has conflicting roles according to roleHierarchy {
            report inconsistency
        }
    }
}
```

Figure 8 **RBAC assessment pseudocode.** Full-size DOI: 10.7717/peerj-cs.376/fig-8

To show the performance of our method on larger systems, the pseudocode for our algorithm is given in Fig. 8. The amount of work necessary scales linearly with both the number of methods in the system and with the product of the REST calls and endpoints within the system, meaning our algorithm runs in $O(M + E \times C)$, where M is the number of methods, E is the number of REST endpoints, and C is the number of REST calls. Since the number of methods in a system is usually much larger than the number of REST calls and endpoints, our algorithm will usually run in $O(M)$. This is in line with the results of our experiment; the discovery module, which searches every method for the needed metadata, was responsible for the majority of the time taken.

THREATS TO VALIDITY

There are several threats to the validity of our work to address. Some of these arise from our experiment and some from how generalizable our approach is.

Internal threats to validity

The primary threats to the validity of our experiment are the accuracy of the violations detected and the accuracy of the performance measures. Since we introduced known mutants for the errors, we know our tool accurately detected all of the issues. Performance-wise, we showed that our tool performed well on a small-sized application, and that the algorithm should scale up well with larger applications since the most expensive portion of the analysis scales only linearly with the number of methods in the project.

External threats to validity

There are three external threats to our work's validity, which may affect how generalizable our results are. First, some of the detected inconsistencies might be false positives that is, those might be intentionally left behind by the developers. Second, it depends on a user-defined role hierarchy that is assumed to contain roles universal to the application. This may not be true if users are defined in separate security realms; a role name in one realm may not be equivalent to the same role name in a different realm, either in its own access rights or in its relative position in the role hierarchy. In this case, a mapping would have to be supplied, showing which, if any, roles should be equivalent across the different realms. Another limitation is the use of security annotations. If security policies are implemented differently than through annotations, are defined in a language or a framework that does not support annotations, the current approach would not detect the roles. However, if another method was used to extract allowed roles, they could be used in the rest of the analysis process.

CONCLUSION

We introduced a novel solution to automatically detect authorization inconsistencies in the RBAC implementation for enterprise applications using automated SAR. Our solution categorizes the violations into five types: missing-role violation, unknown access violation, entity access violation, unrelated access violation, and conflicting hierarchy violation. Our analyzer scans a set of microservice artifacts and provides a report listing all the possible violations by pinpointing their locations and types. While some of the detected violations may be false-positive, the violation type, along with a specific location, helps the developer easily debug them, fix them, or discard them if they were intentional. Although our analyzer was developed for a JAVA enterprise application, our proposed approach is not restricted to any particular programming language or framework. It can easily be implemented for other languages and frameworks since all modern languages now have a well-structured abstraction for REST APIs and RBAC policies.

One major shortcoming of our method is that it assumes the role hierarchy and association of users with roles are defined centrally. However, individual microservices can have separate role hierarchies or even different user-role associations. Similarly, the trust management can be distributed across multiple domains like the dRBAC. In the future, we will extend our system to address these issues to allow multiple role hierarchies and multiple role mappings along with their decentralization. Besides, we like to experiment on role assignment within a user session to identify possible inconsistencies while enforcing DSD. Our long term goal is to perform such analysis within the cloud-native environment commonly used in production deployments, for example, analyzing Dockerfiles and Kubernetes artifacts.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Grant Disclosures

The following grant information was disclosed by the authors:
National Science Foundation: 1854049.
Red Hat Research.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Dipta Das conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.
- Andrew Walker conceived and designed the experiments, analyzed the data, authored or reviewed drafts of the paper, and approved the final draft.
- Vincent Bushong conceived and designed the experiments, analyzed the data, authored or reviewed drafts of the paper, and approved the final draft.
- Jan Svacina conceived and designed the experiments, analyzed the data, authored or reviewed drafts of the paper, and approved the final draft.

- Tomas Cerny conceived and designed the experiments, analyzed the data, authored or reviewed drafts of the paper, and approved the final draft.
- Vashek Matyas conceived and designed the experiments, analyzed the data, authored or reviewed drafts of the paper, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

Data and analysis are available at GitHub, specifically:

- Bytecode analysis: https://github.com/cloudhubs/rad.

- Source code analysis: https://github.com/cloudhubs/rad-source.

- RBAC analysis: https://github.com/cloudhubs/rad-analysis.

REFERENCES

Ahn G-J, Sandhu R. 2000. Role-based authorization constraints specification. *ACM Transactions on Information and System Security* **3(4)**:207–226 DOI 10.1145/382912.382913.

Alshuqayran N, Ali N, Evans R. 2018. Towards micro service architecture recovery: An empirical study. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. Piscataway: IEEE, 47–4709.

Alur D, Malks D, Crupi J, Booch G, Fowler M. 2003. *Core J2EE patterns (core design series): best practices and design strategies*. Second Edition. Santa Clara: Sun Microsystems, Inc.

Basin D, Burri SJ, Karjoth G. 2009. Dynamic enforcement of abstract separation of duty constraints. In: Backes M, Ning P, eds. *Computer Security—ESORICS 2009*. Berlin: Springer, 250–267.

Bass L, Clements P, Kazman R. 2003. *Software architecture in practice*. Boston: Addison-Wesley Professional.

Brachmann E, Dittmann G, Schubert K-D. 2012. Simplified authentication and authorization for restful services in trusted environments. In: De Paoli F, Pimentel E, Zavattaro G, eds. *Service-Oriented and Cloud Computing*. Berlin: Springer, 244–258.

Bruggen DV. 2020. JavaParser: analyse, transform and generate your Java codebase. *Available at https://javaparser.org* (accessed 14 August 2020).

Castillo P, Bernier J, Arenas M, Merelo GuervÃs J, GarcÃa-SÃ¡nchez P. 2011. Soap vs rest: comparing a master-slave ga implementation. *CoRR. ArVix preprint arXiv:1105.4978v1*.

Cicchetti A, Di Ruscio D, Iovino L, Pierantonio A. 2013. Managing the evolution of data-intensive Web applications by model-driven techniques. *Software & Systems Modeling* **12(1)**:53–83 DOI 10.1007/s10270-011-0193-0.

Ciuciu I, Tang Y, Meersman R. 2012. Towards evaluating an ontology-based data matching strategy for retrieval and recommendation of security annotations for business process models. In: Aberer K, Damiani E, Dillon T, eds. *Data-Driven Process Discovery and Analysis*. Berlin: Springer, 103–119.

Ellson J, Gansner E, Koutsofios L, North SC, Woodhull G. 2002. Graphviz—open source graph drawing tools. In: Mutzel P, Jünger M, Leipert S, eds. *Graph Drawing*. Berlin: Springer, 483–484.

Ferraiolo DF, Cugini JA, Kuhn DR. 1995. Role-based access control (RBAC): features and motivations. In: *Proceedings of the 11th Annual Computer Security Applications Conference*. 241–248.

Freudenthal E, Pesin T, Port L, Keenan E, Karamcheti V. 2002. drbac: distributed role-based access control for dynamic coalition environments. In: *Proceedings 22nd International Conference on Distributed Computing Systems.* 411–420.

Granchelli G, Cardarelli M, Di Francesco P, Malavolta I, Iovino L, Di Salle A. 2017. Towards recovering the software architecture of microservice-based systems. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* Piscataway: IEEE, 46–53.

Habib MA, Mahmood N, Shahid M, Aftab MU, Ahmad U, Nadeem Faisal CM. 2014. Permission based implementation of dynamic separation of duty (dsd) in role based access control (rbac). In: *2014 8th International Conference on Signal Processing and Communication Systems (ICSPCS).* 1–10.

Hunsaker C. 2015. REST vs SOAP: when is REST better for web service interfaces? *Available at https://stormpath.com/blog/rest-vs-soap* (accessed 14 August 2020).

Ibrahim A, Bozhinoski S, Pretschner A. 2019. Attack graph generation for microservice architecture. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19.* New York: Association for Computing Machinery, 1235–1242.

JBoss. 2020. Javassist: java bytecode engineering toolkit. *Available at https://www.javassist.org* (accessed 14 August 2020).

Jendrock E, Evans I, Gollapudi D, Haase K, Srivathsa C, Cervera-Navarro R, Markito W. 2014. Working with realms, users, groups, and roles. In: *The Java EE 7 Tutorial.* Vol. 2. Boston: Addison-Wesley Professional.

Jia Y, Harman M. 2011. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* **37(5)**:649–678 DOI 10.1109/TSE.2010.62.

Lee S, Jo J, Kim Y. 2015. Method for secure restful web service. In: *2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS).* 77–81.

McGraw G. 2004. Software security. *IEEE Security & Privacy Magazine* **2(2)**:80–83 DOI 10.1109/MSECP.2004.1281254.

Mohanty H, Mohanty J, Balakrishnan A. 2016. *Trends in software testing.* Singapore: Springer.

Montesi F, Weber J. 2016. Circuit breakers, discovery, and api gateways in microservices. *ArXiv preprint arXiv:1609.05830*.

Mordani R. 2016. JSR 250: common annotations for the JavaTM platform. *Available at https://jcp.org/en/jsr/detail?id=250*.

Oberle D, Eberhart A, Staab S, Volz R. 2004. Developing and managing software components in an ontology-based application server. In: Jacobsen H-A, ed. *Middleware 2004.* Berlin: Springer, 459–477.

Omicini A, Ricci A, Viroli M. 2005. Rbac for organisation and security in an agent coordination infrastructure. *Electronic Notes in Theoretical Computer Science* **128(5)**:65–85.

Oracle. 2020. Securing RESTful web services using Java security annotations. *Available at https://docs.oracle.com/middleware/1212/wls/RESTF/secure-restful-service.htm#RESTF280* (accessed 14 August 2020).

Pallets Projects. 2020. Flask documentation quickstart (1.1.x). *Available at https://flask.palletsprojects.com/en/1.1.x/quickstart* (accessed 14 August 2020).

Quay. 2020. Clair: vulnerability static analysis for containers. GitHub. *Available at https://github.com/quay/clair* (accessed 11 December 2020).

Rademacher F, Sachweh S, Zündorf A. 2020. A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Nurcan S, Reinhartz-Berger I, Soffer P, Zdravkovic J , eds. *Enterprise, Business-Process and Information Systems Modeling*. Cham: Springer International Publishing, 311–326.

Rademacher F, Sorgalla J, Wizenty P, Sachweh S, Zündorf A. 2020. *Graphical and Textual Model-Driven Microservice Development*. Cham: Springer International Publishing, 147–179.

Red Hat Inc. 2020a. Keycloak. *Available at https://www.keycloak.org* (accessed 14 August 2020).

Red Hat Inc. 2020b. Keycloak authorization services guide. *Available at https://www.keycloak.org/docs/latest/authorization_services* (accessed 14 August 2020).

Richards M. 2015. Layered architecture. In: *Software Architecture Patterns*. Newton: O'Reilly Media, Inc.

Salah T, Jamal Zemerly M, Yeun CY, Al-Qutayri M, Al-Hammadi Y. 2016. The evolution of distributed systems towards microservices architecture. In: *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. 318–325.

Sandhu RS. 1990. Separation of duties in computerized information systems. In: *DBSec*. Halifax: Citeseer, 179–190.

Sandhu RS, Coyne EJ, Feinstein HL, Youman CE. 1996. Role-based access control models. *Computer* **29(2)**:38–47 DOI 10.1109/2.485845.

Sandhu RS, Samarati P. 1994. Access control: principle and practice. *IEEE Communications Magazine* **32(9)**:40–48 DOI 10.1109/35.312842.

Scarioni C, Nardone M. 2019. *Pro spring security: securing spring framework 5 and boot 2-based Java applications*. Berlin: Springer.

Son S, Mckinley KS, Shmatikov V. 2013. Fix me up: repairing access-control bugs in web applications. In: *Network and Distributed System Security Symposium*.

Srivastava V, Bond MD, McKinley KS, Shmatikov V. 2011. A security policy oracle: detecting security holes using multiple api implementations. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. New York: Association for Computing Machinery, 343–354.

Steinegger R, Giessler P, Hippchen B, Abeck S. 2017. Overview of a domain-driven design approach to build microservice-based applications. In: *SOFTENG: The Third International Conference on Advances and Trends in Software Engineering*.

Sudhakar A. 2011. Techniques for securing rest. *CA Technology Exchange* **1(3)**:32–40.

Swinhoe D. 2020. The 15 biggest data breaches of the 21st century. *Available at https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html* (accessed 14 August 2020).

Thio L. 2020. Role-based authorization—flask-user v1.0 documentation. *Available at https://flask-user.readthedocs.io/en/latest/authorization.html* (accessed 14 August 2020).

Tihomirovs J, Grabis J. 2016. Comparison of soap and rest based web services using software evaluation metrics. *Information Technology and Management Science* **19(1)**:92–97 DOI 10.1515/itms-2016-0017.

VMware Inc. 2020. Building a RESTful web service. *Available at https://spring.io/guides/gs/rest-service* (accessed 14 August 2020).

Vural H, Koyuncu M, Guney S. 2017. A systematic literature review on microservices. In: Gervasi O, Murgante B, Misra S, Borruso G, Torre CM, Rocha AMA, Taniar D, Apduhan BO, Stankova E, Cuzzocrea A, eds. *Computational Science and Its Applications—ICCSA 2017*. Cham: Springer International Publishing, 203–217.

Wagh DK, Thool R. 2012. A comparative study of soap vs rest web services provisioning techniques for mobile host. *Journal of Information Engineering and Applications* **2**:12–16.

Walker A, Svacina J, Simmons J, Cerny T. 2020. On automated role-based access control assessment in enterprise systems. In: Kim KJ, Kim H-Y, eds. *Information Science and Applications*. Singapore: Springer, 375–385.

Walls C. 2016. *Spring boot in action*. First Edition. Shelter Island: Manning Publications Co.

Xu D, Thomas L, Kent M, Mouelhi T, Le Traon Y. 2012. A model-based approach to automated testing of access control policies. In: *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies, SACMAT^{TM} 12*. New York: Association for Computing Machinery, 209–218.