On Automated Role-Based Access Control Assessment in Enterprise Systems

Andrew Walker, Jan Svacina, Johnathan Simmons and Tomas Cerny

Abstract Software system security gets a lot of attention from the industry for its crucial role in protecting private resources. Typically, users access a system's services via an application programming interface (API). This API must be protected to prevent unauthorized access. One way that developers deal with this challenge is by using role-based access control where each entry point is associated with a set of user roles. However, entry points may use the same methods from lower layers in the application with inconsistent permissions. Currently, developers use integration or penetration testing which demands a lot of effort to test authorization inconsistencies. This paper proposes an automated method to test role-based access control in enterprise applications. Our method verifies inconsistencies within the application using authorization role definitions that are associated with the API entry points. By analyzing the method calls and entity accesses on subsequent layers, inconsistencies across the entire application can be extracted. We demonstrate our solution in a case study and discuss our preliminary results.

Keywords Access control · Microservices · Security · REST · Verification

1 Introduction

With software industry growth the dependency on software engineers increases, which also expands the need to secure systems from outside manipulation. The trends are to move toward the representational state transfer (REST) application programming interfaces (API) [1]. Serving clients with REST-based infrastructures is beneficial in its ease of access by users but can become notoriously complex, particularly when interacting between services. When a system is hard for a user to understand—considering that cogency is an important attribute for software suites [2]—writing functionality and security tests becomes challenging.

A. Walker · J. Svacina · J. Simmons · T. Cerny (✉)
Computer Science, Baylor University, 76798 Waco, TX, USA

A. Walker
e-mail: Andrew_Walker2@baylor.edu

K. J. Kim and H.-Y. Kim (eds.), *Information Science and Applications*,
Lecture Notes in Electrical Engineering 621,
https://doi.org/10.1007/978-981-15-1465-4_38

Testing a system's security is not only a challenge requiring a lot of effort; the confusion surrounding the topic of testing can spawn more vulnerabilities in the system due to false positives and poor coding practices: complexity produces vulnerability [3]. In fact, in a survey conducted by the International Data Group (IDG) in 2014 pertaining to security [3], it was revealed that approximately 63% of applications have not been tested for critical security vulnerabilities or violations. These vulnerabilities can be addressed by implementing standard security features within the system during the development lifecycle [4] itself rather than when it is already too late.

Security vulnerabilities are not cheap either; security violations or breaches can cost companies billions. For example, in 2004, three billion dollars were allocated toward funding security integration. In 2001, loss due to security breaches cost the market over thirteen billion dollars in one year [5]. The discrepancy between preventative measures and asset loss is unfathomable, particularly when considering that such loss was suffered by up to 90% of large corporations and government systems internationally [5]. The National Institute of Standards and Technology found that around $60 billion dollars a year are wasted due to faults in software [6]. Specifically, concerning security violations, an average of $2.1 billion is lost on the market value for large corporations two days after major security breaches which cause outages [6]. Clearly, security vulnerabilities should be taken seriously and dealt with swiftly; the costs involved with a vulnerability found after deployment costs four to eight times more than when they are found and dealt with prior to deployment [6]. Such problems remain challenging as demonstrated by Equifax or Marriott breaches involving hundreds of millions of accounts. Specifically, the total costs for the Equifax incident in 2017 has accrued to over $1.35 billion this year [7].

One of the methods of securing software, particularly with REST APIs, is by using Role-Based Access Control (RBAC); wherein each user in the system has one or multiple roles assigned that grant defined privileges into the system [8]. Roles are applied in enterprise systems via multiple ways such as by annotations, by configuration files, etc. This paper considers the annotation-based role restrictions. A well-defined security role policy would protect from many possible vulnerabilities by minimizing overlap in privilege between roles so no single leak in privilege could cause disaster to the suite [8]. However, writing a bullet-proof security definition is no easy task, particularly when attempting to create one for a complex work environment with hundreds of employees all needing access to disparaging components in a unified system. There is no documentation or methodology for consistently writing tests for a system that guarantees useful information, and similarly, writing security definitions for a system is equally ambiguous and relative to the needs and environment established [9]. Therefore, there is an apparent need to continually verify the security of a system automatically to help developers with the arduous process of securing their software.

Being able to iteratively and automatically verify security roles applied inside of a REST API would be of remarkable benefit, not only for reducing downtime when errors are retroactively found but for also lowering financial costs by reducing the risk of break-ins down the line and subsequently the need for hiring specialized

penetration testers. Other cost reduction factors, found by research studies [10], noted that using static code analysis to discover security vulnerabilities can reduce production costs up to potentially even 23%. Static code analysis is also beneficial in that the same research also showed an increase of discovered security vulnerabilities over previous methods by 2.6 times [10]. Furthermore, using penetration testing over static analysis techniques can become intrusive, causing an accidental denial of service attacks to the system in question.

System administrators should be wise when choosing which methodology of testing they wish to implement when considering the intrusiveness and potential risks attributed with penetration testing and the fact that a static analysis tool is only as powerful as its model it is based on [10]. However, with the benefits outlined above, our motivations for developing a static analysis tool are evident.

The experiment presented below focuses on a single proprietary microservice named QMS, developed separately but re-purposed here as a benchmark for our static analysis tool. QMS belongs to a family of microservices [11], however, the static analysis tool provided only highlights the intra-microservice analysis whereas the future research goal will include inter-microservice analysis including the other microservices in the related family and the interactions therein. This paper will introduce the subject of static analysis on a codebase with a provided security role hierarchy and provide a list of potential or immediate security role violations and vulnerabilities.

The paper is organized as follows. Section 2 describes related work. The proposed method is outlined in Sect. 3, followed by a case study in Sect. 4. Finally, we conclude the paper and describe our goals for future work and list the references.

2 Related Work

When it comes to API development, the two main choices are REST and SOAP (Simple Object Access Protocol) [12]. What should be clarified is that while REST is an architecture for API development, SOAP is merely a protocol. The prolific nature of these two approaches toward API development draws comparison due to their particular tendencies to evolve a system toward specific structures regardless of their semantic differences. Major advantages toward REST include how easy it is as a developer to learn, REST messages are lightweight and can run effectively on mobile, REST calls are based on the standard hypertext transfer protocol (HTTP), and parsing JavaScript Object Notation (JSON) is faster than parsing Extensible Markup Language (XML) [13], making REST faster than SOAP [14].

REST was the most popular web API technology in 2014 with 69% being written in REST [13]. Current corporations that use the REST architecture or have implementations of it available include Docker and OpenStack [15].

Securing data is generally easy in REST, which is important given how important security is described above. Several methods of securing a REST API include separating code into separate packages and only revealing relevant entry points, using

prefixes and writing styles with enumeration to reveal only the methods which a user should have access to, and writing metadata into the code in particular ways to restrict access such as with annotations [16]. Organizing and enumerating code for this purpose is both bulky and fragile, and certain metadata methodologies can cause this kind of code murkiness [16]. Thus, as many developers choose to do in Java, we shall focus on using annotations as metadata for security through Java EE Security API [17].

REST APIs with RBAC are structured in such a way that users with given roles only have access to such methods that their roles are intended to provide. Thus, given a hierarchy such that there is an administrator role and a user role below that, the administrator should be able to access any behavior and data that the user has access to, however, the user should not be able to interact with the data and methods exclusively available to the administrator.

Recent Java EE Security API Specification [17] recommends the use of annotations to restrict access on each API endpoint of the application. Using Java annotations, it could appear as follows: @*AllowedRole* administrator [16] With numerous options to declare restrictions. In REST API annotations, such configurations in the metadata exist over methods desiring access control. The use of annotations is the easiest and most reasonable way.

Some APIs may prefer the use of custom annotations, such as @*AdminRole* which is a sufficient replacement for the previous example—or so that something like @*CustomerMethod* could be used to describe a method that is for customers explicit use [16]. We will focus on the Java EE Security API annotations. Custom annotations would not affect our benchmark they represent an abstraction.

Due to the prolific nature of security role issues, plenty of research and development has gone into the areas surrounding role violations or tool-assisted security role development. One such tool developed and described by Ciuciu et al. [18] was implemented in order to help developers get recommendations of appropriate security annotations based on the context via a large ontology created from provided business information. Since the tool works independently from any source code, if the business knowledge provided is faulty then the recommendations provided by the tool will also be faulty. All that both this oracle and our tool can do is make sure the security roles represented in code and those defined by the developer are cooperative.

Similar research to our own includes the development of a security violation finding oracle that intends for access control within an API system [19]. The major difference with our research is that they not only are currently capable of analyzing multiple microservices at once, but they also require more than one in order for the oracle to find discrepancies at all [19]. This inter-microservice analysis style is a future target for our research as a whole, however, our success in intra-microservice analysis stands on its own. Another benefit to our system over this oracle is faster static analysis [19] and thus our tool could be run much more frequently on a codebase.

A similar research study [20] described a method of statically analyzing RBAC for the criteria of consistency, completeness, and redundancy. It checked whether an access control rule set is consistent across the methods, covers all subsets of

permissions, and whether permissions are unnecessarily repeating permissions in subsets of methods. The coverage of a hierarchy with respect to access control over a set of methods is not necessarily always correlated with security. However, if a method has RBAC defined but incomplete ad preventing user from making an action, it is more of a flaw with the system than a potential security breach. Our idea of completeness reflects more on the topic of security: whether a method has RBAC when it should.

The FixMeUp tool [21] is a static analyzer for access control testing, however, the main differences being that their focus is on PHP instead of Java and that FixMeUp edits the code as well as analyzes. As intriguing as it is to edit the code automatically, this introduces potential syntax errors or unintended effects on the methods in question. Though both our tool and FixMeUp could be prone to false positives, our tool could not affect the codebase negatively as it never modifies the code it is observing.

3 Proposed Method

Enterprise applications are often structured into three distinct layers—the controller layer, the services layer, and the repository layer. Some commonly known terminology may denote the layers as the presentation layer, business layer, persistence layer, and database layer. In this case, presentation would be the controller as they both represent the frontend of the API for the users. The business layer handles the requests, so it is referred to here as the service layer. Lastly, the persistence layer contains the data access object logic for interaction with the database layer. For generalization, both layers are condensed and referred to here as the repository layer [2, 22]. In an RBAC enterprise application, each endpoint is secured with required roles through annotations [16], configuration or some other system. At the controller layer, where the endpoints are defined, the authorization is easy to control. Configuration in an RBAC system will focus on security for the endpoints, with defined roles for each, however, often will not define roles for functions in lower layers. Due to this lack of explicit control, as you move into the lower layers, the authorization becomes less clear. This potential tangling of roles renders current authorization checks insufficient.

We propose a static analysis method to identify areas of inconsistencies in role access definition, especially with access to data. These areas are identified using a hierarchy of roles given by the operator. These roles are structured into a tree, with the highest permission role as the root. This role tree must be prepared before the use of our system and must be prepared by the operator since the roles in the hierarchy must match those used in the enterprise system. Using the structure of the tree, the roles are organized into a distinctive ordering and it becomes trivial to see when a lower access role is conflicting with a higher access role. Below we will go through our process for aggregating the violations from the examination of the enterprise application.

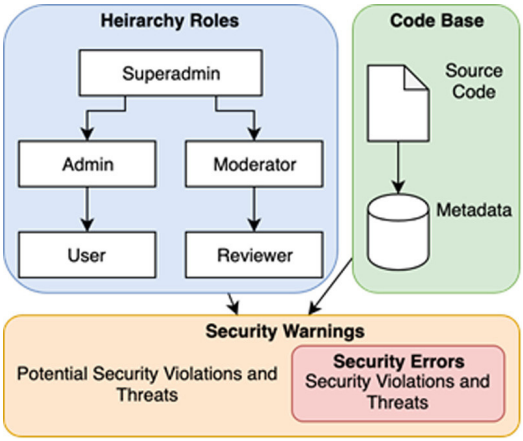

Fig. 1. A model to aggregate security roles and code metadata to discover violations

Our proposed method is divided into two phases, the discovery phase and the weaving phase. The discovery phase introspects the relevant security meta-data from the application itself. The metadata is then used to create consistent discrete structures representing each endpoint for use in the next phase. The weaving phase aggregates the discrete security structures to discover the violations in the application. In addition to the metadata from the codebase, this phase also uses the role hierarchy tree (see Fig. 1). By dividing the analysis into two phases, the amount of information needed to be introspected from the system can be reduced and more complex calculations can occur in the weaving phase instead of the discovery phase which can slow down the introspection process.

The discovery phase of our system extracts the needed information from the system and encapsulates it into manageable structures for analysis. At this point, the scraped information is discrete instead of holistic. The bulk of the information is extracted from the REST endpoints of the system since the endpoints are the places in which the user can directly access the system and also serve as the establishment points for security roles within the system.

Each endpoint is introspected to discover the security roles associated with it. Most enterprise systems use annotation for this [16]. This information, along with later tracing of the method call flow is enough to find most violations in the application. For the remaining violations, more information from each method is needed. First, the HTTP type of the endpoint is extracted into the security structure. The endpoints can be one of the main HTTP types (GET, POST, PUT, etc.) [23]. Additionally, our system stores the parameter lists and return types of each method. This security structure is associated with the endpoint it was extracted from and the collection of these structures is passed onto the next phase. At this point, our prototype is limited to the Java platform, however, easy to transfer to other platforms that use RBAC.

The weaving phase receives the collection of discrete security structures representing each endpoint in the system. Additionally, the weaving phase takes the role tree from the user which it will use to discover the violations. First, the system checks each endpoint for any unknown roles and missing security definitions. Once those violations have been found, each endpoint structure is then used as the starting point

to recursively traverse down its method flow, annotating each sub-method with its parent's security roles.

If an *endpoint* has multiple security roles associated with it, only the lowest role is passed to its children. If a *non-endpoint* has multiple security roles associated with it, then all of them are passed down to its child. After all methods have been annotated, the system searches for any methods with multiple security roles associated with it. Any method with multiple security roles is considered to be a violation and the system uses the hierarchical role tree to distinguish between the types of violations. Lastly, the system checks each endpoint for duplication of HTTP type, parameter list and return type with conflicting roles. The system stores those violations with the others and a collection of all violations found in the application is presented to the user.

The violations that our system will discover fall into one of five categories that we have defined: hierarchy access violations, unrelated access violations, entity access violations, unknown role violations, and missing security definition violations. These violations are discovered as the larger security context is aggregated from the security structures provided from the discovery phase. The interpretation of each violation is done using the role tree to understand the semantics of the security roles. This allows our system to determine when parts of the system are accessed by conflicting roles.

The necessity of classifying the violations in the enterprise systems is rooted in making violations of a potentially very large and complex system more manageable to developers. These violations must be ultimately interpreted by the developers of the system to determine if they are valid concerns or not. Organizing the violations into categories helps the developers analyze them easier.

Using these violations, our proposed method covers all possible data access leaks found in RBAC enterprise applications. Our system will discover unrestricted and inconsistent data exposure as well as data exposure in a request with insufficient permissions according to the business specification. Below we will walk through the errors found in a benchmark enterprise application.

4 Case Study

The Question Management System (QMS) enterprise application was developed at Baylor University as part of an NSF grant proposal for Central Texas Computational Thinking, Coding and Tinkering. The application was built using Spring Boot [24] with defined controller, service and repository layers and handles authorization using pure RBAC with annotations on each endpoint of the API. The hierarchy of roles for the QMS application can be seen in Fig. 2. In a blind study, a single mutant [25] for each type of violation was introduced into the application which caused 10 total violations. Our system was able to successfully locate all of the violations through analysis of the mutated application. Below we discuss more details on the analysis process go through the violations found in the mutated application.

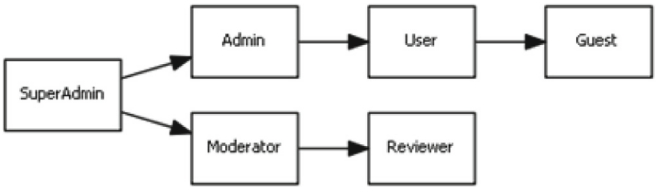

Fig. 2. Role hierarchy for a testbed

In Spring Boot applications, the layers of the application are defined with annotations *(@Controller*, *@Service*, and *@Repository*). For our purposes, we looked for a subtype of @Controller, *@RestController*, which denoted where the endpoints of the API were located. The HTTP type was extracted from either *@RequestMappting* with a given parameter or from a subtype (*@GetMapping*, *@PostMapping*, etc.). The parameter lists and return types were also extracted from each endpoint. For extracting the security roles associated with each endpoint, we look for *@RolesAllowed* annotations and their parameters that list allowed roles for a given endpoint, which we extract and associate with each endpoint security structure. Each endpoint is instrumented to find the control-flow starting at the endpoint tracing through the service and repository layers.

A *hierarchy violation* occurs when a part of the system is accessed by two directly conflicting roles. We classify directly conflicting roles as the situation when one of the roles is an ancestor of the other role. The application mutant [25] introduced for this violation type was to create inconsistent roles for creating and deleting Category objects. This mutant caused hierarchy access violations in the repository layer of the application. Our system located the mutant and correctly identified the violating sub-method. Figure 3 visualizes the control flow graph that caused the violation. Interpreting hierarchy access violations is more complex than the other types due to inherent ambiguity. With RBAC it is impossible to tell if a lower role is accessing a higher privilege flow which would be a violation or if a higher role is accessing a lower privilege flow which would not be a violation. Due to these limitations, our system is unable to distinguish a potential hierarchy access violation from an actual one.

In addition to the hierarchy access violations, our system identified *unrelated access* violations stemming from a mutant. An unrelated access violation is very similar to a hierarchy violation but occurs when two roles that are not directly linked both access the same parts of a system. In the QMS system, the unrelated access violation occurred when a moderator and user had access to the same method in the repository layer. The method flows are similar to Fig. 3 hierarchy violation, however

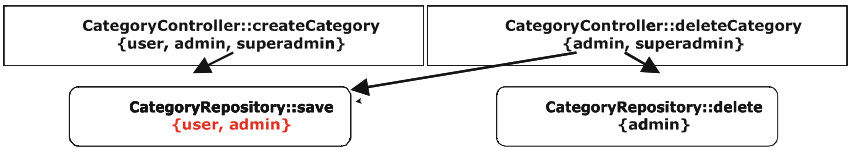

Fig. 3. Control flow graph of a hierarchy access violation

neither role is a direct ancestor of the other. This violation highlights parts of the system where overlapping concerns may be present, and refactoring could be needed.

Additionally, our system successfully identified an *entity access* violation stemming from a mutant. This violation is slightly different than the preceding violations as an entity access violation deals with access to operations on a specific entity. A violation here occurs if two endpoints with the same HTTP type, parameter list, and return value have differing security roles. This is a violation since the system is granting similar access to two different roles. Whereas the other violations focus on the sub-system below the endpoints, this violation deals directly with the endpoints and the access to the system. The QMS system had two endpoints, *updateCategory* and *updateCategoryName*, both in the *CategoryController*. Each of these endpoints had the same HTTP type, parameter list, and return type. The reason for the violation is that the mutant granted the up-date *Category* endpoint admin access rights whereas the *updateCategoryName* has user access rights. In the case of an entity access violation, the differing roles are enough to constitute a violation. That is to say that it doesn't matter to the system whether the conflicting roles are hierarchically related.

There are two more types of violations introduced into QMS which our system successfully located. An *unknown role* violation occurs when a role is found on an endpoint but is not found in the role tree. This usually happens in one of two cases. The first case is due to the misspelling of the role in the endpoint definition or role tree. The second is the accidental exclusion of the role from the role tree. In the case of QMS, a mutant was introduced into an endpoint to misspell the "user" role as "uder". The mutant affected 4 total methods, the initial endpoint and the 3 methods it called. Our system identified this as an unknown role and flagged all the children methods as vulnerable. Lastly, our system will flag any *exposed endpoint with no security definition*. With security definitions often being added at the end of development [3] it is easy to overlook an endpoint which leaves resources unprotected. Our system will warn the user when an endpoint is found without security definitions and flag all sub-methods of the endpoint as vulnerable. A mutant was introduced into QMS which removed the security constraints on a single endpoint. This mutant affected 3 total methods, the initial endpoint and the 2 child methods it called. Our system successfully located the violation and flagged all 3 methods as vulnerable.

5 Conclusion

We proposed a novel solution for automatic role-based access control testing in enterprise systems and proved the validity of the solution on the benchmark example. Our method is based on static code analysis and traces the roles through-out the control-flow graphs. It is able to detect the following security threats: hierarchy access violations, unrelated access violations, entity access violations, unknown role violations, and missing security definition violations. By preventing these violations, we restrict unauthorized users or users with insufficient permission from accessing resources within the system. Though the testbed was using Java annotations, our

tool would not be restricted to this use case given time to develop new ways to scrape access controller methods and other parsing tools for analyzing security access violations in different programming languages.

In the future, our system will account for a multitude of ways to extract security definitions. Our future work will include integration with Git repositories to check only the methods that require re-scanning via the commit differences. In this way, we can make our system faster and more harmonious with other continuous integration processes. A microservice application deals with security consistency within itself and with the other modules in the system. We will extend the introspection to include inter-microservice flows. Our long-term goal is to develop this sort of complex and dynamic feature.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 1854049.

References

1. Vural H, Koyuncu M, Guney S (2017) A systematic literature review on microservices. In: Computational science and its applications—ICCSA 2017, pp 203–217. Springer, Cham
2. Steinegger R, Giessler P, Hippchen B, Abeck S (2017) Overview of a domain-driven design approach to build microservice-based applications
3. AnwerMohd F, Mustafa N (2016) Security testing. Trends in software testing
4. McGraw G (2004) Software security. EEE Secur Priv 2:80–83. https://doi.org/10.1109/MSECP.2004.1281254
5. Mercuri RT (2003) Analyzing security costs. Commun ACM 46(6)
6. Telang R, Wattal S (2007) An empirical analysis of the impact of software vulnerability announcements on firm stock price. IEEE Trans Softw Eng 33(8):544–557. https://doi.org/10.1109/TSE.2007.70712
7. Schwartz MJ (2019) Equifax's data breach costs hit $1.4 billion. https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473
8. Dinh KKQ, Truong A (2019) Automated security analysis of authorization policies with contextual information. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-58808-6˙5
9. Felderer M, Zech P, Breu R, Bchler M, Pretschner A (2016) Model-based security testing: a taxonomy and systematic classification. Softw Test Verif Reliab 26(2):119–148. https://doi.org/10.1002/stvr.1580
10. Bardas AG (2010) Static code analysis. RomIan Econ Bus Rev 4(2):99–107. https://ideas.repec.org/a/rau/journl/v4y2010i2p99-107.html
11. Cerny T, Donahoo MJ, Trnka M (2018) Contextual understanding of microservice architecture: current and future directions. SIGAPP Appl Comput Rev 17(4):29–45. https://doi.org/10.1145/3183628.3183631
12. Tihomirovs J, Grabis J (2016) Comparison of soap and rest based web services using software evaluation metrics. Inf Technol Manage Sci 19(1):92–97. https://doi.org/10.1515/itms-2016-0017
13. Levin G (2015) The rise of rest API. https://blog.restcase.com/
14. Aihkisalo T, Paaso T (2012) Latencies of service invocation and processing of the rest and soap web service interfaces. In: 2012 IEEE eighth world congress on services. pp 100–107. https://doi.org/10.1109/SERVICES.2012.55

15. Li L, Chou W, Zhou W, Luo M (2016) Design patterns and extensibility of rest API for networking applications. IEEE Trans Netw Serv Manage 13(1):154–167. https://doi.org/10.1109/TNSM.2016.2516946
16. Bodkin R (2004) Enterprise security aspects
17. Will Hopkins AT (2017) Java EE security API specification (jsr 375). https://javaee.github.io/security-spec/
18. 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. pp 103–119. Springer, Cham
19. 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. pp 343–354. PLDI '11, ACM, New York, USA. https://doi.org/10.1145/1993498.1993539
20. 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, pp 209–218. SACMAT'12, ACM, New York, USA. https://doi.org/10.1145/2295136.2295173
21. Son S, Mckinley KS, Shmatikov V (2013) Fix me up: repairing access-control bugs in web applications. In: Network and distributed system security symposium
22. Richards M (2015) Software architecture patterns. O'Reilly Media, Inc
23. Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, Berners-Lee T (1999) Hypertext transfer protocol. https://tools.ietf.org/html/rfc2616
24. Software P (2019) Spring framework. https://spring.io/
25. Jia Y, Harman M (2010) An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng 37(5):649–678