Towards Automated Inter-Service Authorization for Microservice Applications

Conference Paper · August 2019

DOI: 10.1145/3342280.3342288

3 authors, including:

Xing Li
Zhejiang University
7 PUBLICATIONS **61** CITATIONS

SEE PROFILE

Towards Automated Inter-Service Authorization for Microservice Applications

Xing Li
Zhejiang University

Yan Chen
Northwestern University

Zhiqiang Lin
The Ohio State University

CCS CONCEPTS

• **Security and privacy → Authorization**; • **Software and its engineering → Cloud computing**.

KEYWORDS

Microservice, Policy-Based Access Control, Automation

ACM Reference Format:
Xing Li, Yan Chen, and Zhiqiang Lin. 2019. Towards Automated Inter-Service Authorization for Microservice Applications. In *SIGCOMM '19: ACM SIGCOMM 2019 Conference (SIGCOMM Posters and Demos '19), August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3342280.3342288

1 INTRODUCTION

As an emerging software architecture, the microservice idea divides traditional monolithic software into multiple *microservices* according to its business boundaries. Microservices communicate with each other through lightweight network API invocations (e.g., HTTP, gRPC). Each of them can be independently developed, deployed, and upgraded, which greatly improves the flexibility of software development and scaling. Benefiting from the recent booming of container technologies, microservice has been widely used as the basic architecture of modern cloud applications.

However, in this environment, communications between microservices are exposed through the network, which creates a potential attack surface. It is unrealistic to rely solely on network boundaries to provide full security protection. When a microservice is compromised, it can send malicious requests to other microservices to initiate attacks or steal data. Therefore, in addition to using SSL/TLS to protect the communications among microservices, popular microservice infrastructures such as *Kubernetes* [1] and *Istio* [2] also provide inter-service authorization mechanisms to specify which invocations a microservice can initiate. For example, a compromised *logging* service might talk to a *backend* service to get sensitive information, and the administrator can specify that only the *frontend* service can access the *backend* service to defend against this attack.

In order to achieve high control flexibility, the inter-service authorization mechanisms usually use complex access control policies to perform fine-grained authorization. Currently, however, this still relies on the careful manual configuration by the administrator, which is error-prone and tedious. Besides that, due to the large scale of modern microservice applications[1], it is unrealistic for administrators to manually configure and maintain access control policies for every microservice. To make matters worse, microservice's dynamic nature of frequent iterations requires the access control policies to be updated accordingly in time, which is also an "impossible mission" for administrators. Therefore, to make a powerful access control mechanism really work here, automation is essential, otherwise it is just a castle in the air.

Solution	C	F	A
Document-based approaches [3, 4]	✗	✗	✓
History-based approaches [5]	✗	✓	✗
Jarvis (proposed approach)	✓	✓	✓

Table 1: Comparison of Jarvis with existing works.

The automation of inter-service authorization requires the automated generation, maintenance, and update of access control policies. An ideal solution is expected to fulfill the following goals, which face respective challenges:

- **Completeness (C).** To generate reasonable access control policies, it needs to automatically extract the invocation logics between microservices from the application's business logic. The challenge is how to ensure the completeness of logic extraction, since microservices may be developed in different languages and interact in diverse ways.
- **Fine Granularity (F).** To generate fine-grained access control policies, in addition to the dependencies among microservices, it also needs to mine the detailed attributes of the inter-service invocations. This is also challenging because these attributes are varied and usually not explicit.
- **Agility (A).** Given the dynamic nature of microservices, it needs to timely perceive the changes in microservice invocation logics and dynamically adjust the access control policies. The challenge is how to quickly infer and incrementally update the access control policies in large scale scenarios.

Unfortunately, these goals can hardly be jointly addressed by merely adopting existing security policy automation studies. The recent studies utilize machine learning methodologies to mine and extract reasonable security policies from documents [3, 4] or historical operation data [5]. However, as demonstrated in Table 1, the **document-based** approaches are usually not fine-grained and not complete enough (e.g., Text2Policy [4] achieved an average recall of 89.4%); the **history-based** approaches relies heavily on the quality of historical data, and only sufficient historical data can lead to complete security policies, which limits their agility.

Inter-service authorization should follow the business logic of microservice applications and meet the principle of least privilege.

[1]Twitter has $O(10^3)$ microservices in 2016. (http://bit.ly/twitter-linuxcon)

完整性：提取逻辑

细粒度的访问控制
：挖掘服务调用间的详细属

敏感性：及时发觉微服务调用逻辑的变化，
并动态调整访问控制策略

SIGCOMM Posters and Demos '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-6886-5/19/08…$15.00
https://doi.org/10.1145/3342280.3342288

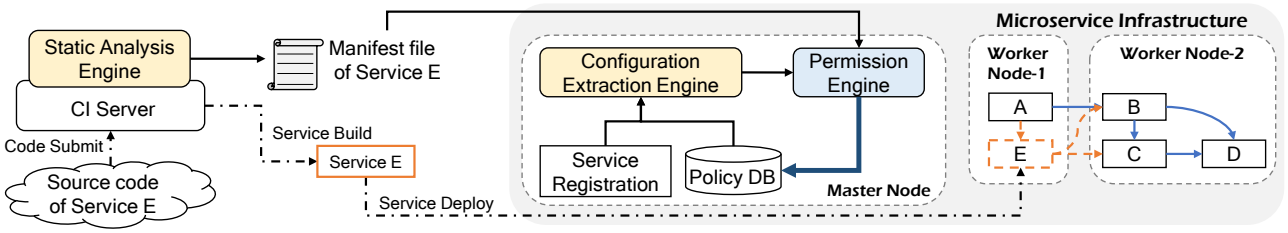

Figure 1: The architecture and workflow of Jarvis.

Thanks to mature code review mechanisms, the code of microservices is a trusted material that can accurately reflect their business logics. Therefore, when a microservice initiates a request that is **not** in its code to other microservices or external network, we consider the request to violate the business logic, that is, the microservice is compromised and the request is malicious. Based on this insight, we propose **Jarvis** as the first automated inter-service authorization mechanism for microservice applications. Seamlessly integrated with microservice lifecycle and infrastructure, it automatically extracts the possible invocations a microservice may initiate by static analysis, and then generates access control policies with information such as service registration at deployment time. Besides that, Jarvis monitors the changes of microservices and quickly adjusts corresponding access control policies when the changes occur.

2 SYSTEM DESIGN

As shown in Figure 1, the architecture of Jarvis includes an offline *Static Analysis Engine*, an online *Configuration Extraction Engine* and an online *Permission Engine* located on the master node of the microservice infrastructure. Next, we will describe the system design of Jarvis from the three steps of its workflow.

2.1 Request Extraction

To obtain complete dependencies among microservices at a fine-grained level, it is necessary to obtain detailed information about API invocations (e.g., *URL* and *method* for HTTP calls). To this end, Jarvis uses a program-slice based static analysis mechanism to extract requests and related attributes from microservices developed in different languages.

The dotted line in Figure 1 depicts the standard CI/CD[2] process for microservice E. At first, the source code of E is submitted to the CI server, where a series of tools run for automated testing and code checking. The *Static Analysis Engine* is deployed here to extract the invocations that E may initiate and generate a manifest file to describe them. Microservices use a limited number of protocols to communicate, such as HTTP, gRPC, etc. Therefore, to narrow down the state space and accelerate the static analysis, Jarvis first scans the code of E along the control flow and identifies the statements that make network API invocations, such as `requests.get()`. Next, it uses these as starting points to perform backward taint propagation on the control flow to get the program slice related to each request. Finally, Jarvis extracts useful attributes, such as *URL*, *method*, etc., from the slices by semantic analysis, and represents indeterminate fields that depend on the input of upstream requests with wildcards.

[2]Continuous integration & continuous deployment, an automatic workflow for microservice development and deployment.

2.2 Policy Generation

To generate fine-grained access control policies, we take all information that reflects the invocation relationships among microservices into account. There may be instances of multiple versions of the same microservice in the system, and they may have different manifest files. We process them separately and distinguish them by tags in the access control policy. Besides the manifest file describing what invocations a microservice may make according to its code logic, Jarvis identifies which microservices provide the interfaces it calls by the actual service registration information, that is, finds the callees. In addition, inter-service traffic management rules can also affect the fine-grained authorization. For example, if the administrator specifies service B to send all requests with feature f to service D and the rest to service C, a request q with f should not be able to access C. Hence, Jarvis also extracts this kind of rules from the *Policy DB* of the infrastructure. With all the data sources mentioned above, the *Permission Engine* eventually generates access control policies, aggregates them for management and subsequent updates, and deploys them into the *Policy DB*. The following policy enforcement will be performed by the microservice infrastructure.

2.3 Policy Update

The deployment, upgrade, and policy distribution of microservices are all achieved by the microservice infrastructure and corresponding configuration files (e.g., deploy.yaml). Therefore, Jarvis monitors the infrastructure and uses pruning to quickly deduce whether the change will affect inter-service authorization when the microservice application changes. If so, it uses a dynamic update mechanism to efficiently calculate the adjustments that need to be made on the access control policies and incrementally update them as needed for large scale microservice applications.

3 PRELIMINARY RESULTS

We evaluated the static analysis mechanism with the 3 most popular open-source microservice demos[3], which consist of 22 microservices developed in 6 languages. Jarvis extracted 37 HTTP requests, 20 gRPC requests, and 39 TCP requests from these microservices, which achieved 100% coverage over the ground truth obtained by manual analysis. The result shows the completeness of Jarvis, which is the basis for automated authorization.

ACKNOWLEDGMENT

This work is supported by National Key R&D Program of China (2017YFB0801703) and the Key Research and Development Program of Zhejiang Province (2018C01088).

[3]**Bookinfo**: http://bit.ly/bookInfo, **Hipster Shop**: http://bit.ly/hipstershop, and **Sock Shop**: http://bit.ly/sock-shop.

REFERENCES

[1] The Linux Foundation. Production-Grade Container Orchestration - Kubernetes, 2019. Accessed on 2019-6-30.

[2] Istio. Istio: Connect, secure, control, and observe services., 2019. Accessed on 2019-6-30.

[3] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. A deep learning approach for extracting attributes of abac policies. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 137–148. ACM, 2018.

[4] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 12. ACM, 2012.

[5] Leila Karimi and James Joshi. An unsupervised learning based approach for mining attribute based access control policies. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1427–1436. IEEE, 2018.