Khazar Journal of Science and Technology
Khazar University Press 2022
Volume 6 №2 2022, 85-89
DOI: 10.5782/2520-613.202.6.2.85

Security Analysis While Transitioning from Monolithic Applications to Microservices

Tural Sadigov

Graduate School of Science, Art and Technology,
Khazar University, Azerbaijan
Corresponding author: tural.sadiqov@khazar.org

Abstract

Microservice architectures have evolved as an enticing alternative to more typical monolithic software application approaches. Microservices give various benefits in terms of code base knowledge, deployment, testability, and scalability. As the information technology (IT) industry expands, it makes sense for IT behemoths to adopt the microservice, but new software solutions creates new security vulnerabilities, as the technology is young and the faults have not been adequately mapped out. Authentication and authorization are key components of any software with a significant number of users. However, owing to the lack of microservice research, which derives from their relatively young, there are no specified design standards for how authentication and authorization are best performed in a microservice.

This thesis analizes existing microservice in order to safeguard it using a security design pattern for authentication and authorization. Different security patterns were assessed and different degrees of security helped in identifying an acceptable security vs. performance trade-off.The objective was to strengthen the patterns' validity as known security patterns. Another purpose was to establish a security pattern that was suitable for the microservice.

Introduction

Usually in backend applications, often referred as monolithic applications, the code is developed and deployed as a whole, single project. But in microservices, this

artifact is divided into multiple small applications or services that can be developed, tested and deployed independently from each other. Today most companies are trying to shift from monolithic to microservices because of its effective approach to development, but in the current microservices DevOps environment, there are new and evolving challenges for developers and teams to consider on top of the more traditional ones.

As it becomes much more difficult to maintain a microservices setup than a monolithic one, each microservice setup may evolve from a wide variety of frameworks and coding languages and this brings new challenges to the development environment and security is the top one to consider.

Authorization and authentication are the foundations of security for every application, monolithic or not. The MSA offers potential improvements at many stages of the software development process, but it also creates new challenges. Unsurprisingly, when multiple components of an MSA need to communicate with one another, securing requests to and from as well as within a microservice becomes a much more difficult task than it is for a monolithic application, where authentication and authorization can be done once when accessing the application.

The main goal of the research is look into the security patterns that may be used to organize authentication and authorization in a microservice to implement a security solution.

Background

The concept of microservices has been known since the early 2000s, however the name "microservices" only appeared in particular situations in the early 2010s (Richardson, 2019). However, others claim it was coined as late as 2014 (Zimmermann, 2017). Still, the concept of microservices is very new when compared to other software development methods and architectures. Microservices can be thought as different small applications independent of each other (in reality no services are fully independent though), so it makes the whole application more reliable, because if some of the services is not available, it does not affect the others. Since small services becomes large in number, Kubernetes can be used to orchestrate the whole application by running each service in individual virtual machines. Figure 1 describes MSA and communication flow of microservices.

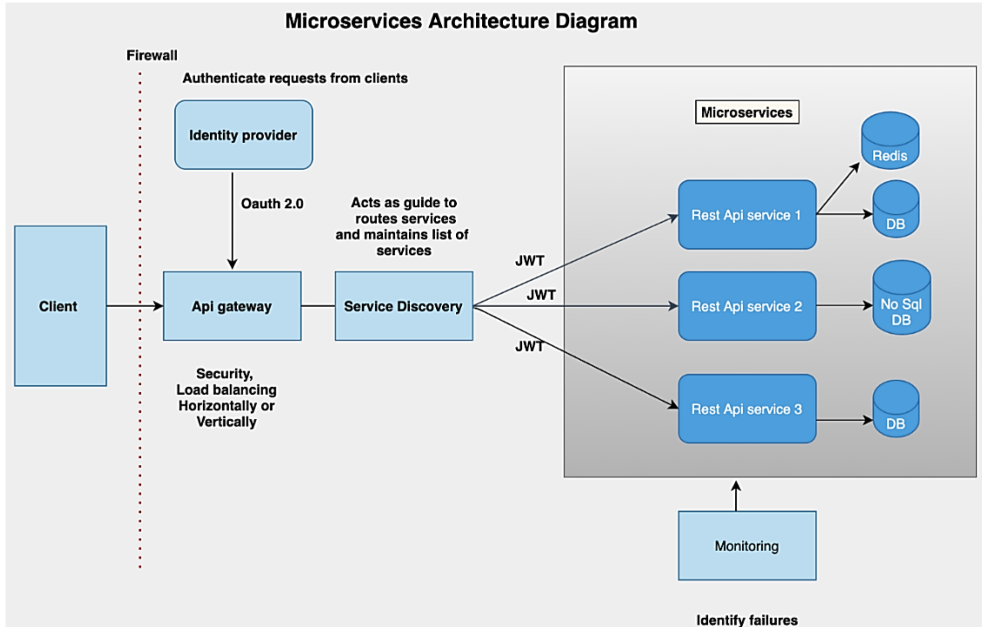

Figure 1. Graph displaying a microservice and communication flows

So, as seen in the figure, each service can communicate with each other through API gateway. API gateway is a middleman between services and external client requests.

IPC is the protocol used to communicate between services and the API gateway. Hypertext Transfer Protocol (HTTP) and, by extension, Hypertext Transfer Protocol Secure are two implementations of such a mechanism (HTTPS). Because the HTTP protocol is stateless, there is no built-in mechanism for a server to remember any interactions with a client. In order to safeguard resources, future HTTP requests must remember a previously authenticated and approved client in case they need to be reauthenticated and reauthorized. To preserve the verified status, a token comprising user information and permissions may be supplied with each subsequent request. The JSON Web Token is one such standard (JWT). It may be used to transfer information in the form of a JSON object, which can then be signed or encrypted to guarantee integrity or confidentiality. In their paper (Xu, 2019), Rongxu Xu, Wenquan Jin, and Dohyeun Kim propose how an MSA may be protected using JWT. It is anticipated in this technique that an API Gateway intercepts all requests so that an authorization server may give JWTs for future requests to sensitive data services.

Moreover, OAuth 2.0 is used to secure the microservices. OAuth is an open standard

which minimizes the number of permission stages by requesting a user to give a service authorization to other services holding sensitive data (OAuth, 2022). Today, the OAuth protocol is regarded outdated since its successor version accomplishes the same function but has minimal technical similarities (Hardt, 2012).

Authentication and authorization in a Kubernetes microservice

This chapter dives into the technical aspects of the authentication and authorization components' implementation. The authentication and authorization service (abbreviated auth-service) were required, along with a Redis store, gateways, and some example services (which emulates the business logic of a microservice).

Kubernetes is used to implement the microservice. Because all traffic inside the cluster is inaccessible from the outside, an ingress controller allows communication into the cluster from the outside (i.e., the internet or the local network in which the cluster is implemented). The microservice is deployed in a Kubernetes cluster that employs an NGINX ingress controller variation. The service that is to be exposed (in this case, an edge level gateway) will be assigned an ingress object (which defines how the ingress controller should route traffic related to the service) that specifies a reachable URL if the requests come from a device connected to the internet or an internal network.

The auth-service may authenticate users by interacting with an LDAP server, which also replies with a user's roles. By providing a JWT to a logged-in user, this token may be simply utilized to both identify a user for authentication and locate the related roles of this user for authorization.

The services are implemented in Spring Boot and provide straightforward REST endpoints that may either return a value directly or trigger another call to another microservice to get further resources.

The implementation of the three distinct authentication and authorization security patterns—edge level (3.1), service group (3.2), and service level (3.3) gateway patterns—is covered in depth in this section

Edge level gateway pattern

The simplest of the three security patterns is the edge level gateway pattern. Despite offering the least level of protection, it was shown to be the most popular method of establishing authentication and authorization in a microservice. As a result, it may

also be used as a benchmark against which to evaluate the other two designs (as both are more complex and provides a higher level of security). Without initially submitting a request to the edge API gateway, none of the services offered by the microservice are accessible to clients or servers outside of the microservice.

Service group gateway pattern

The service group gateway design extends the edge level gateway pattern by grouping together services that need the same degree of access to access. The auth-service also handles this authorization. An example would be a collection of services that all need the same role to access. This subset of services, just like the edge gateway, would be secured by an additional gateway that is likewise located behind the edge gateway. This adds an extra layer of security. Another feature that is comparable to the edge level gateway approach is that communication that does not need to transit through a gateway is not subject to authentication or authorization. This implies that services behind the same internal gateway may send requests without being authenticated or authorized

Service level gateway pattern

The service level gateway pattern is the third and final pattern examined in this thesis. This security architecture necessitates that each service has its own gateway that secures it through authentication and permission. Because it is not always possible to have one role linked with one service, some or all of the gateways might have the same role necessary to provide access to the protected service. While it may seem that role verification is unneeded when two service-gateway pairs interact and require the same role to access, it really offers a unique type of security. Because all connections inside the microservice between its services need a security check, services controlled by a bad actor cannot reach any other conceivable targets without first undergoing authentication and authorization

The testing framework

This chapter discusses the two types of tests that are conducted. Section 4.1 discusses the research methodology that influenced the testing technique. Section 4.2 covers the security testing procedure. Finally, section 4.3 goes into depth on the load testing that produced the most of the findings.

Research process for load testing

The load testing software (JMeter) was used to analyze the findings and provide numbers such as the median and average. Additional Python programs were used to analyze the data. The Python programs employed linear regression to display the data trend such that a forecast for how larger loads than what was tested may result could be made. The scripts were also used to generate visual representations of the data, such as scatter plots and box plots, in order to provide a broader variety of data.

Security tests

- The security tests examined four situations that may occur during normal use. These were:
- Using the microservice with valid credentials
- Omitting the authorization token
- Sending an invalid token
- Sending a valid token to a user who does not have the appropriate role to access the requested resource

Load testing

Using JMeter load testing, the three distinct security schemes' effects on response times were evaluated. The real loads and service should preferably be as similar to the tested service as feasible for accurate load testing, meaning that the load generator's delay should be as low as possible. As a result, a Spring Boot application specifically designed for testing was installed on the machine being tested, making all communication within the cluster local. Figure 2 shows a graphic depiction of the location of the JMeter-running service and the flow of requests to the microservice.

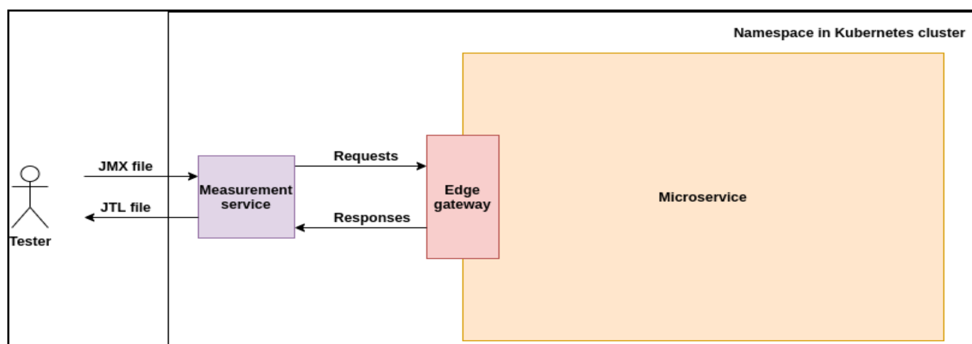

Figure 2. Graph displaying the tester and measurement service in relation to the microservice being tested

Results and disscussions

Three situations were evaluated to see whether the auth-service and gateways provide the necessary protection.

It was crucial to make sure that the security solution also rejected requests that were found invalid since the load testing mostly focused on valid requests. The situations listed in Section 4.2 were put to the test to confirm this. Since the token is legitimate and the corresponding user has all necessary responsibilities for permission. When a no or incorrect token is received, an error message with a cause is sent.

The findings of the testing are shown in this section using box plots and scatter charts. As the number of threads rises across all security types, they show a consistent rise in response times. What is particularly notable is that as the load grows, the gap in reaction times between the security patterns widens. It seems that the service level gateway pattern is more significantly impacted than the service group gateway design. Comparing the service level gateway pattern to the other two security patterns, this shows a quicker increase rate in response times.

For simpler comparison, Table 5.1 presents all median values. Response times for the edge level gateway pattern increased from 1225 milliseconds for a single thread to 2362 milliseconds for 2000 threads. This represents a growth of roughly 93%. The comparable increase for the service group gateway pattern was 1244 milliseconds to 3086 milliseconds, or a 148% increase. Last but not least, the increase for the service level gateway pattern was 1260ms to 4367ms, or a 247% percentage increase. There was a 31% rise from edge level to service group, an 85% increase from edge level to service level, and lastly a 42% increase from service group to service level when comparing the percentage increases of the 2000 threads load across the three security models. All percentages were rounded to the nearest integer, as you will see.

Table 1. Median response times for the security patterns

Threads	Edge level gateway response time (ms)	Service group gateway response time (ms)	Service level gateway response time (ms)
1	1225	1244	1260
100	1243	1284	1315
200	1248	1306	1332
300	1272	1323	1371
400	1370	1487	1473
500	1516	1719	2013
600	1555	1687	2049
700	1644	1815	2094
800	1697	1765	2536

900	1715	2007	2639
1000	1752	2115	2687
1100	1854	2274	2740
1200	1937	2201	3065
1300	2006	2124	3292
1400	2026	2526	3447
1500	2047	2595	3388
1600	2067	2625	3750
1700	2207	2374	3892
1800	2181	2901	4017
1900	2292	2925	4145
2000	2362	3086	4367

Conclusion

It is necessary to establish what is considered a favorable outcome before recommending a security pattern. The natural logic would be to seek a balance between strong security and fast reaction times. However, even if a security pattern has longer reaction times than the other patterns, it may still be considered to be doing well in comparison to a fair benchmark. When seeking to secure a system, it is frequently preferable to utilize technology that has been shown to be capable of delivering the desired security. This concept guided the selection of technologies for the security patterns, which resulted in the use of JWT, role-based authorization, and API gateways.

In terms of performance against security, the service group gateway pattern is the best option. There is no reason not to propose the security pattern with the highest level of protection, as the findings indicate that the impact on performance will be almost equivalent to that of the design with the lowest level of securityAs a result, the service level gateway design is the preferred security pattern for the project's microservice.

Acknowledgments

The load tests in this thesis were inspired by the performance testing in Akhan Akbulut and Harry G. Perros's work (2019). Also The recommended solution in the paper was one of the key sources of inspiration for what would become the most basic layer of security: the edge level gateway design. Finally, great thanks to the Google, which provided free 3 month trial to use all services that Gcloud offers.

References

Akbulut, A., Harry G. (2019). Performance Analysis of Microservice Design Patterns, IEEE Internet Computing. Vol. 23, pp. 25, 69.

Hardt, D. (2012). The OAuth 2.0 Authorization Framework. Available: http://www.rfc-editor.org/rfc/ rfc6749.txt.

OAuth Core 1.0 Revision A. (2022). Available: https://oauth.net/core/1.0a/.

Richardson, S. (2019). Microservice Patterns: With Examples in Java. Manning Publications, pp. 2, 10, 12, 13, 18, 2019.

Xu, W. J. (2019). Microservice Security Agent Based On API Gateway in Edge Computing. Sensors, 19 (22)

Zimmermann, O. (2017). Microservices tenets. Computer Science-Research and Development. vol. 32, no. 3, p. 10