Applying Spring Security Framework and OAuth2 To Protect Microservice Architecture API

Quy Nguyen*, Oras Baker
Southern Institute of Technology, Invercargill, New Zealand.

* Corresponding author: Email: Oras.baker@sit.ac.nz
Manuscript submitted March 1, 2019; accepted May 15, 2019.
doi: 10.17706/jsw.14.6. 257-264

Abstract: Since 2014, Microservice Architecture (MSA) has been widely applied and deployed by big companies such as Google, Netflix and Twitter. This is a way of architecting software systems in which the services of a single application are decomposed then deployed and executed separately. This research examines the possibility of applying Spring Security Framework and OAuth2 to secure microservice APIs which are built on top of Spring Framework. By developing a Proof of Concept (POC) of an Inventory Management System using MSA on top of Spring Framework, Spring Security Framework and OAuth2. we have conducted security tests over the POC using unit testing and manual testing techniques to examine if there are any vulnerabilities and we were able to show and confirm the effectiveness of the Spring Security Framework and OAuth2 in securing Spring-based APIs.

Key words: Software architecture, microservice architecture (MSA), oauth2, spring framework, and POC.

1. Introduction

The traditional Monolithic approach of software architecture requires the entire application stack to be bundled together for each deployment. This concept creates many drawbacks for the application, especially the inflexible scalability, the high cost of resources and refactoring effort, difficulties of the DevOps between distributed teams [1]. Microservice Architecture (MSA) is supposed to address these problems by decomposing the application into separated services; each service takes responsibility for a single business capability and is deployed and executed independently.

Application communicate with each other via the network communication protocols and the Internet, so that this architectural style heavily depends on the Application Programming Interfaces (API). Given that, APIs in a microservice application are required to be appropriately secured to protect the application and its resources against the threats that deal with API invocations.

The aim of this research is to reduce the knowledge gap on MSA and API security by developing a Proof of Concept (POC) of an MSA application using Spring Framework, Spring Security, and OAuth2, then performs security testing using Unit Testing and Manual Testing techniques over the POC.

2. Background and Literature Review

Since the very first assessments by enterprises for the effectiveness and the impact of MSA to enterprises by 2012 [2], interest in MSA has significantly increased over the recent years according to Google Trends statistic [3]. MSA are being implemented by big companies to scale their applications in the cloud in an

efficient way, to reduce complexity, to quickly expand development teams and to achieve agility [4]-[6]. Netflix, Amazon, and SoundCloud are just some of the big firms that have adopted MSA for their enterprise and web applications and deliver their services to all over the world [7], [8].

Regardless of the vital role of the API security in MSA, the literature review shows that the studies that focus on MSA at API endpoint level are just a few. There is a study conducted by Salibindla (2018) on Microservice API security; however, this study focused on security for the communication protocols and did not provide implementation guide for any specific language. Xie, Han *et al.* (2017) [10], also performed a study on the design and implement of Spring Security. Nevertheless, these studies were conducted separately, and there exists no study that confirms the effectiveness of Spring Framework (SF), Spring Security Framework (SSF), and OAuth 2.0 (OAuth2) when these technologies are applied to implement authentication and authorisation for MSA API endpoint.

2.1. Microservice Architecture and Microservice API

In contrary with the traditional monolithic software architecture where the whole application with many services inside is deployed together within the same application server, MSA refers to the applications architecture that decomposes an application into a set of narrowly focused, single responsibility, lightweight and independently deployable services, a.k.a. microservices [10]. These services can communicate with each other using APIs [9] The two major communication protocols and data exchange formats for microservice API are Simple Object Access Protocol (SOAP) and Representational State Transfer (REST). Due to the lightweight nature of REST, this research uses REST for the API implementation and experiment.

MSA does not make an application any simpler, it only distributes the application logic into multiple smaller components. Although the decomposition brings many benefits such as the scalability, high availability, it is, however, resulting in a much more complex network interaction model between components especially when the application is made of too many services [11], [12]. Fig. 1 presents a comparison between the monolithic and microservice architectural style of a simple online shopping system.

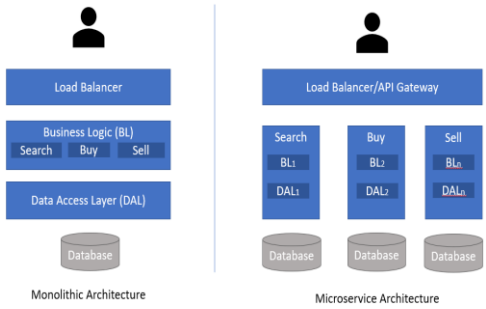

Fig. 1. A comparison of monolithic architecture and an MSA.

3. Proof of Concept Development

The POC is built to answer the research questions. Moreover, it should reflect the applications of MSA in a real business context. Therefore, a POC of an inventory management system is an appropriate experimental application. As introduced in Fig. 4, OAuth2 not only is applied for the web-based application but also is applied to the backend services without a web browser and/or user interaction. Thus, the POC should be designed in a way that supports the security testing for both of these implementation types. Moreover, the experiment should also examine OAuth 2.0 for both authentication and authorisation purposes. In

accordant to the experimental requirements, key features and actors are proposed in the next subsections.

3.1. Use Cases

3.1.1 Use cases for authorisation server

The Authorisation Server plays the role of IdP in OAuth2 workflow. The prototype provides the fundamental use cases of the Authorisation Server as shown in Fig. 2. The two actors who interact with the Authorisation Server are:

- Resource Owner: The resource owner in OAuth2 workflow.
- Client Apps: Is the client application that is registered with the authorisation server.

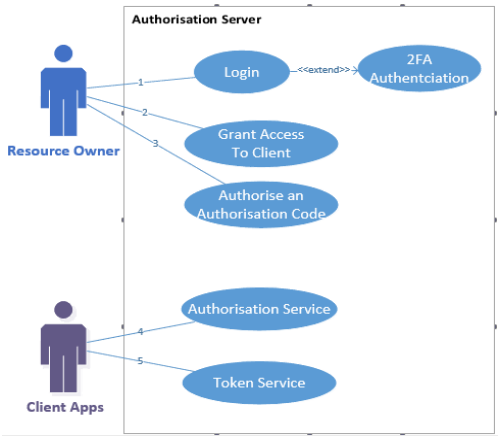

Fig. 2. Authorisation server use cases.

3.1.2 Use cases for resource servers

The POC implements two microservices which take responsible for managing the mobile phone and jewellery respectively. The microservices play the role of the RPs in the OAuth workflow. Use cases of the resource servers are depicted in Fig. 3.

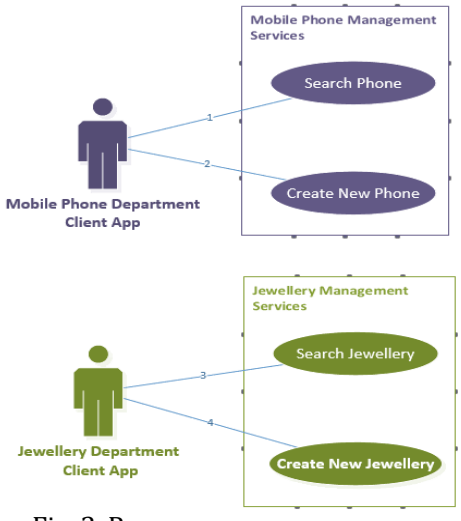

Fig. 3. Resource server use cases.

4. High Level Architecture

Fig. 4 introduces the High-Level Architecture (HLA) of the Proof of Concept. This HLA presents the core

modules and dependencies at the framework level that involves in the implementation of the POC.

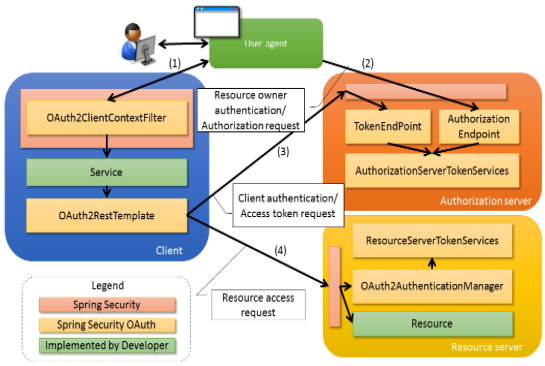

Fig. 4. Application high level architecture.

a) Client requests authorisation for the resource owner. As shown in the above Fig. 4, the request is directly made to the resource owner.
b) The client receives an authorisation grant (described later) as the credentials representing authorisation from resource owner.
c) Client requests for an access token by presenting own authentication information and authorisation grant given by resource owner to the authorisation server.
d) Authorisation server authenticates the client and confirms the validity of the authorization grant. It issues the access token if the authorisation grant is valid.
e) Client requests to the resource protected on the resource server and authenticates with the issued access token.
f) Resource server checks the validity of access token and accepts the request if it is valid.

5. POC Development

5.1. Authentication Service

The function of the Authentication Service is to request information from an authenticating user or client and validate it against the configured identity repository using the SSF implementation.

5.2. Security Configuration

The application security configuration extends the built-in WebSecurityConfigurationAdaper of Spring Security Framework, the application security configuration provides the key features below:

a) userDetailsService: look up user details from the database for authentication purpose
b) BCryptPasswordEncoder ensure that password is encrypted using BCrypt cryptographic.
c) The session attribute "_csrf" protect the application from CSRF attack.
d) Http request is required to be authenticated.

5.3. OAuth2 Authorisation Configuration

OAuth configuration extends the built-in AuthorizationServerConfigurerAdapter of Spring Security Framework with the additional implementation for OAuth2 authorisation support, the OAuth2 configuration provides the key features below:

a) oauthDataSource: looks up client detail from the database for the authorisation process
b) TokenStore: the access tokens are also stored in the database and accessed using Java Database Connectivity (JDBC) technique.
c) ApprovalStore: approval information is stored in the database and accessed using JDBC technique.

d) AuthorizationCodeServices: similar to the ApprovalStore, authorisation codes are stored in DB.
e) configure(ClientDetailsServiceConfigurer clients): Configure the ClientDetailsService, declaring individual clients and their properties.
f) void configure(AuthorizationServerEndpointsConfigurer endpoints): Configure of the Authorisation Server endpoints such as a token store, authentication code service, token customisations, user approvals and grant types.

5.4. Account Service

Account service takes responsibility for returning user information from a given a token.

6. Experiments and Findings

6.1. Experiments

This section presents the conducted experiments over the POC including CSRF attack, XSS attack, and Brute Force attack at the API endpoints. We used Spring Mock MVC test library to simulate the real attacking parameters and allows for repetition if needed.

6.1.1. Experiment 1 — CSRF attack

CSRF Attack Test 1 – CSRF Protection is disabled in WebSecurityConfig

In this experiment, Spring Security CSRF Protection is disabled so that the API is not protected from CSRF attack, and an authorised client tries to create a new mobile phone without providing CSRF token.

CSRF Attack Test 2 – CSRF Protection is disabled in WebSecurityConfig

In this experiment, the CSRF is disabled in the WebSecurityConfig. As such, the API is not protected by Spring Security CSRF Protection. Follow the CSRF configuration; this research experiments with the case an unauthorised client tries to create a new mobile phone without providing CSRF token.

CSRF Attack Test 3 – CSRF Protection is enabled in WebSecurityConfig

In this experiment, CSRF configuration is enabled by the default configuration of the SF. With this configuration, the experiment simulates the case an authorised client tries to create a new mobile phone without providing CSRF token.

CSRF Attack Test 4 – CSRF Protection is enabled in WebSecurityConfig

In this experiment, CSRF configuration is enabled by the SF default configuration. After that, the experiment simulates the case of an authorised client tries to create a new mobile phone without a valid CSRF token. Data set up and the HTTP status code recorded are shown in Table 1.

Table 1. CSRF Attack Test 4 — CSRF Enabled – Authorised Client – Valid CSRF Token

Experiment 3c	Parameter Name	Value
Input Parameters	WebSecurityConfig – csrf enabled	True – CSRF is enabled
	Username	john
	Password	123
	Csrf token	Is randomly generated by the Spring Security Framework
Collected Data	Http status code	201 - CREATED

6.1.2. Experiment 2 – XSS attack

In this experiment, the client initiates a POST request to create new product and examines if the response is XSS protection enabled. Data set up and the response headers recorded as shown in Table 2.

Table 2. XSS Protection Experiment Setup

Experiment 2a	Parameter Name	Value
Input Parameters	Service Endpoint	http://localhost:8081/product
	HTTP Verb	POST
	Product Info	Name: LG Phone, SKU: 5897
Collected Data	Response Header: X-XSS-Protection	1
	Response Header: mode	block

6.1.3. Experiment 3 – Brute force attack

In this experiment, this research uses Spring MVC Test framework to simulate the case attacker using password dictionary to perform Brute Force Attack. Data set up and the response headers recorded. The Unit Test for Brute Force attack is implemented.

Brute Force Attack Test 1 – Normal failure login – No Brute Force detected.

In this test, this research performs two continuous login failures, which is still in the threshold of maximum attempt allowed and simulates a normal case of user forgot username and password.

Brute Force Attack Test 2 – Continuous Login Failure that exceeds the maximum attempt allowed

In this test, this research performs three continuous login failures, which exceeds the maximum login attempt allowed, and simulates a normal case of Brute Force attack detected.

7. Findings

In the experimental process, this research focuses on testing the POC against the common vulnerabilities that are related to microservice API endpoints. These vulnerabilities include CSRF attack, XSS Attack, and Brute Force attack. There are four testing scenarios performed to reflect the attacks described by OWASP. In all scenario, security implementation and configuration in the POC has successfully protected the API Endpoints from the attack with 100% test cases passed. In case of Brute Force attack detected, the client account is disabled to prevent further attacking using password dictionary. This finding is the evidence to confirm that the combination of SSF and OAuth2 protects SF-based API endpoint from common API vulnerabilities as shown in Table 3.

Table 3. Summary of Hacking and Penetration Experiment

Hacking and Penetration Test Scenario	Test case	Passed (Yes/No)
CSRF Protection Disabled	Access with a valid credential	Yes
	Access with an invalid credential	Yes
CSRF Protection Enabled	Access with a valid credential, invalid CSRF token	Yes
	Access with an valid credential, invalid CSRF token	Yes
XSS Attack	Validate a response header with XSS attack protection	Yes
Brute Force Attack	Normal Login Failure	Yes
	Brute Force Attack Detected	Yes
Total Pass Rate		100%

Finally, we were able to design and develop the recommended software architecture together with the implementation guide for the POC in Section 4 Even though the POC is just a simplified version of an

Inventory Management System, it still covers the essential features of a typical MSA application with OAuth2 authorisation standard.

8. Conclusion and Future Work

In an MSA application, microservices communicate with each other through the service API endpoints. In other words, an API endpoint is a location where the services can access and attain the resources they need to perform their function. The API endpoint plays a vital role in guaranteeing the correct functioning of the services and systems that interact with it and is the interface where the data are exchanged between services. Thus, security for API endpoint is one of the key security factors in an MSA application. The review of existing literature and related works found that there exists no experimental result to confirm the effectiveness of SF, SSF, and OAuth2 when these frameworks and security standard are used to secure the API endpoints in an MSA architecture. Therefore, this research implemented a POC of an inventory management system using MSA as an experimental system to examine the integration between the technologies. The experimental results show that the POC successfully protected the resources from the CSRF attack, XSS attack and Brute Force attack. Moreover, the POC disabled the user's account in case of Brute Force attack detected. Thus, it is evident that SSF and OAuth2 protects SF-based API endpoint from common API vulnerabilities. As a continuation of this research, in the future, the researcher wishes to extend the research to cover the security for the whole API implementations, including the security for other application layers such as business layer, data access layer. As such, suggest a more comprehensive API security solution for the Java-based microservice application.

References

[1] A., K., Torkura, I.H., M., Sukmana, & Meinel, C. (2017). Integrating continuous security assessments in microservices and cloud native applications. *Proceedings of the10th International Conference on Utility and Cloud Computing* (pp. 171-180). Austin, Texas, USA: ACM. doi:10.1145/3147213.3147229

[2] ThoughtWorks. (2014). *Microservices.* Retrieved August 21, 20018, from Technology Radar: https://www.thoughtworks.com/radar/techniques/microservices

[3] Google. (2018, Aug 01). *Google Trends - Web Search interest:* Microservices. Retrieved August 21, 20018 from Google Trends: https://www.google.com/trends/explore?hl=en#q=microservices

[4] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *Proceedings of the 2015 10th Computing Colombian Conference*.

[5] Kritikos, K., & Massonet, P. (2016). An integrated meta-model for cloud application security modelling. *Procedia Computer Science*, 84-93.

[6] Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, motivations and Issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, *4(5)*, 22-32.

[7] Alshuqayran, N., Ali, N., & Evans, R. (2016). A systematic mapping study in microservice architecture. *Proceedings of the 2016 IEEE 9th International Conference Service-Oriented Computing and Applications (SOCA)* (pp. 44-51).

[8] Harms, H., Rogowski, C., & Iacono, L. L. (2017). Guidelines for adopting frontend architectures and patterns in microservices-based systems. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (pp. 902-907). Paderborn, Germany: ACM.

[9] Salibindla, J. (2018). Microservices API security. *International Journal of Engineering Research & Technology*, *7(1)*, 277-281.

[10] Xie, L., Han, L., Li, M.-H., & Dong, X.-L. (2017). Design and implement of spring security-based T-RBAC.

Proceedings of the 2017 International Conference on Wireless Communications, Networking and Applications (pp. 183-188). Shenzhen, China: ACM.

[11] Sun, Y., Nanda, S., & Jaeger, T. (2015). Security-as-a-service for microservices-based cloud applications. *Proceedings of the 2015 IEEE 7th International Conference Cloud Computing Technology and Science* (pp. 50-57).

[12] Syed, M. H., & Fernandez, E. B. (2017). The container manager pattern. *Proceedings of the 22nd European Conference on Pattern Languages of Programs* (pp. 1-9). Irsee, Germany: ACM. doi:10.1145/3147704.3147735

Oras Baker received his PhD in artificial intelligence from the University of Malaya in 2009. He is currently the head of School of Computing and the programme manager for postgraduate studies and master of IT at the Southern Institute of Technology, Invercargill, New Zealand. His research interests include artificial intelligence, web intelligence, virtual and augmented reality techniques, data mining, IOT, and energy efficiency.