Optimization of Microservices Security

D.C Kalubowila
Department of Computer Science and Software Engineering
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
dckalubowila25132@gmail.com

S.M Athukorala
Department of Computer Science and Software Engineering
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
sachin2262716@gmail.com

B.A.S Tharaka
Department of Computer Science and Software Engineering
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
tharakabas@gmail.com

H.W.Y.R Samarasekara
Department of Computer Science and Software Engineering
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
yasiru97@gmail.com

Udara Srimath S. Samaratunge Arachchilage
Department of Computer Science and Software Engineering
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
udara.s@sliit.lk

Dharshana Kasthurirathna
Department of Computer Science and Software Engineering
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
dharshna.k@sliit.lk

2021 3rd International Conference on Advancements in Computing (ICAC) | 978-1-6654-0862-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICAC54203.2021.9671131

***Abstract*— Microservices is a trending architecture, and due to its demanding features and behaviors, billions of business applications are developed based on it. Due to its remarkable ability to deploy and coordinate containerized microservices, Kubernetes deployments support the service mesh architectures, and that ensures secured inter-service communication. The Istio is the widely used service mesh tool at present. However, service-to-service communication happens in the present Istio architecture, and there is a probability of exchanging unauthorized and over-provisioned requests due to incorrect implementation. Currently, these requests are verified within the upstream microservice. Obtaining a response to an erroneous request may take considerable time latency. This research thereby aims to address a solution to reduce the latency of a response by implementing an external validation model. The proposed external validation model ensures that the required parameters are validated and actions are taken before requests reach the service level. External validation enables applications to save significant time and resources.**

Keywords—Kubernetes, Istio, Microservices, Reverse Proxy, Envoy, Inter Service Communication security, Pod security, Optimizaton

I. INTRODUCTION

During a symposium on Cloud Computing in 2005, Dr. Peter Rogers used the term "micro web services" for the first time. The concept "microservices" was originally introduced in 2011 to describe a specific implementation of Service-Oriented Architecture (SOA) [1], which is a typical architectural technique for breaking down complex systems into tiny, loosely coupled services. The core concept of the microservice architecture is divide and conquer, which implies splitting down a larger task into smaller, more manageable concerns that are implemented independently. With the advent of microservices architecture, the best practices of DevOps and Cloud computing have been adopted by companies like Amazon [2], Netflix [3], Facebook [4], and Uber [5] to develop robust applications [6],[7].

The ability to effectively package microservices by encapsulating all necessary libraries and dependencies made containerization a much convenient approach. Because of services like Docker, containerization has become the favored method for efficiently deploying microservices [8]. However, in an enterprise-level application, deploying microservices is a complex task when the number of microservices increases. In response to this problem, the Kubernetes orchestration framework [9] was introduced by Google in 2014 to enable companies to operate more robust distributed systems by offering effective solutions for load balancing, replication, auto-scaling, and service discovery. In addition to other qualitative attributes, security is one of the most challenging tasks in a microservices architecture. Unlike monolith, microservices have multiple endpoints to the application. Each microservice should have its entry point [10]. In a Kubernetes cluster, a service mesh architecture is utilized to secure inter-service communication. A service mesh could be described as an infrastructure layer that manages inter-service communication [11]. Istio is one of the widely used service mesh in Kubernetes. Istio's integration with Kubernetes enables complicated deployments to benefit from standardized, universal traffic management, telemetry, and security. The proposed research introduces a model to reduce network latency due to malicious requests in the Istio service mesh. The primary components of the proposed approach are as follows:

- Proxy level auth configuration and throttling methodlogy outside from the microservice business infarstructure.

978-1-6654-0862-2/21/$31.00 ©2021 IEEE

- Provide the required security matrices based on code level criticality analysis from each programmable microservice.
- Mechanism to capture and remove over provisioned scopes in a request token, outside from the business infrastructure.
- Analyze errors in service-to-service communication with microservices tracing.

The content of this publication is divided into the following sections. Section II outlines the context and background work with the literature references consulted in developing this security optimization model. Section III describes the technique used to build the proposed model and a high-level summary of its core components. Section IV analyzes the results acquired from implementing the constructed model, and finally, Section V summarizes this publication's conclusion and prospects for research.

II. Background and Literature Review

Kubernetes' current approach, auth policies are managed at the edge through an API gateway. However, when service to service communication occurs, there is a possibility of exchanging unauthorized/over-provisioned requests because of erroneous implementations. These requests are currently validated within the upstream microservice. However, getting a response to an illegitimate request takes some time. The inability to provide the subject feature of this concern negatively impacts the deployment of large-scale microservices.

OAuth system could allow object transactions using plain text tokens. To mitigate the harm caused by such token over scoping attacks, [12] recommends limiting the scope of an access token. This paper misses an opportunity to propose a solution for the Transport Layer Security(TLS). The research [13] showcases a SaaS-specific Role-Based Access Controll (RBAC) architecture. The research conducted by Balamurugan [14] suggested a cloud Token Management System (TMS) . This research is accomplished by a behavior-based token generation mechanism for the user's cloud resource operations. It raises certain worries about fraudulent processes that some legitimate users obtain a valid token from the TMS. The TMS should expand beyond creation and logging to include resource-based access control modeling and administration Several articles have emphasized the need of improved security models in microservices along with some of the general issues in existing microservices architecture in cloud hosted Kubernetes clusters. Melton et al. [15] discussed the importance of a reverse proxy sidecar container in each Kubernetes pod that offers network-level security without requiring any modifications to the upstream microservices [15]. Once Istio installed all inter-services communication between Docker containers secured with Mutual TLS (MTLS)[12].

They defined the gap in the research by implementing a large-scale vulnerability detection method utilizing Machine Learning in the research publication [24]. As with the research [25], that team also developed a completely automated machine learning approach for analyzing source code for security vulnerabilities. Additionally, they attempted to identify sources, sinks, validators, and methods of authentication. Both approaches focused mostly on code vulnerability.

There are numerous commercial and open-source solutions for microservice tracing and root cause analysis, as well as prior research. Google Dapper, a Large-Scale Distributed Systems Tracing Infrastructure [17] is the most recent sophisticated research on distributed tracing. This study provides the foundation for the majority of solutions and concepts. Most Commercial and open-source product are given metrices and traces for developers to make easy to debug their application. Although there are open standards, APIs, and instrumentations for distributed tracing such as Open Tracing [18]. The sidecar pattern links a secondary container to the primary parent container to share the same lifecycle and resources. Burns et al. [19] discussed both pros and cons of the sidecar pattern. In contrast, the sidecar design simplifies microservice code by abstracting away from conventional infrastructure-related work and ensures that the application's underlying platform is loosely coupled. The increased inter-process communication latency is considered a disadvantage.

III. Methodology

Although performance is not a security characteristic, it has been a significant influence in determining whether security measures are adopted or rejected .in reality. The proposed model comprises four major components: 1) Middleware at the proxy level, 2) Token manager, 3) Code analyzer and 4) Trace analyzer. Each is interconnected, depicted in Fig 1. The following subsections give an in-depth analysis of the methods used to build the proposed approach, as well as an overview of the components' respective functionality.

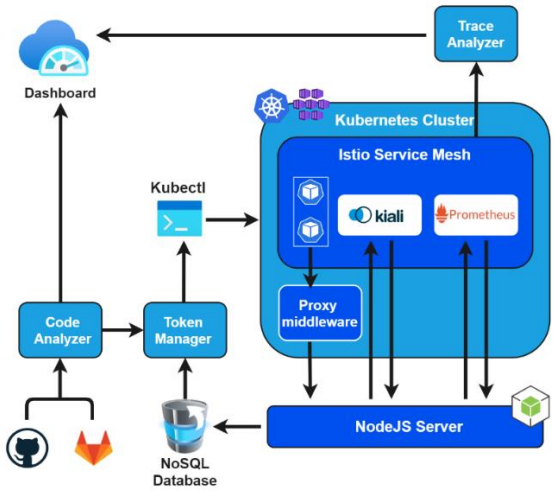

Fig. 1. High level diagram of the proposed model

A. *Proxy level auth and rate limitting middlewear*

The Table I highlight the 03 sub-tasks consisted of the proposed proxy level middleware.

TABLE I. SUB TASKS OF THE PROXY MIDDLEWARE

Sub Task	Description
Auth Configuration	Evaluates the rights of the request towards the upstream microservice.
Access Token extractor	Capture metadata of access token from the reverse proxy level and provide to token manager component.
Rate limiting controller	Used to limit the number of requests made during an outage.

Envoy is used as the Istio reverse proxy. Once the Istio service mesh is installed, each Kubernetes pod will be secured by Envoy reverse proxy [20]. In this middleware approach, envoy reverse proxy is extended using a Web Assembly (WASM) filter. Creating a native WASM Envoy filter is time-consuming due to the increased memory consumption required to run one or more WASM virtual machines. In this research, the middleware mentioned above was created using Envoy Proxy WASM Software Development Kit(SDK), which executes WASM filters inside a stack-based virtual machine, isolating the filter's memory from the host environment. The SDK has implementations from various programming languages like C++, Rust, Assembly Script, and Go [21]. In the research, the Rust SDK has been used by the authors. Only the Envoy can identify WASM extensions. The above Envoy extension has been divided in 02 components as follows:

- Singleton delegator – Forward request metadata to the middleware.
- Cache Filter - Periodically access updated cache details from the middleware.

In the proposed middleware approach, the reverse proxy will function in the way shown in Fig. 2. In the absence of sufficient cache storage inside the proxy, the cache is backed up asynchronously by the middleware to an outside storge through an API. If an unauthorized, unauthenticated, or rate limit exceeded request is sent towards an upstream microservice, the proxy level enhancement will instantly block and alert, according to the established method in Fig 3, Previously, once a request is sent to an upstream service, the microservice verifies its legitimacy. If database access is required to determine the legitimacy of a malicious request, the database access latency will also have a detrimental effect on the response.

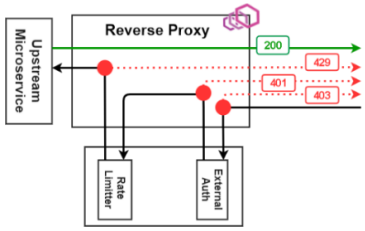

Fig. 2. Vulnerable request detection from the Envoy extension

According to the placement of the above enchaned proxy level middleware solution, the authors have introduced 03 new external authentication and authorization configuration methodologies as follows. Strict Auth can be used to enable fine grained access control while Peer Auth can be used to cros-grained access controll.

TABLE II. DISCOVERED NEW AUTH METHODOLOGIES

Auth Methods	Located
Sidecar Auth (Pod level)	Inside the K8 pod and Istio data plane
Strict Auth	Outside the K8pod and Istio control plane
Peer Auth	Outside the K8 pod and Istio control plane

B. *Token Manager*

The token Manager component is a part of the newly implemented proxy level middleware. Token manager consists of three subcomponents depicted in Table III.

TABLE III. SUB TASKS OF THE TOKEN MANAGER

Sub Component	Description
Token Analyzer	Capture metadata and generates inputs for policy generators with code analyzer metrics.
Policy Generator	Generate proxy policy according to inputs from the token analyzer.
Control Agent	Enforce policy and generate metrics.

To fulfill the mentioned analyzing process, it needs code level security metrics and the captured access tokens provided by the code analyzer and proxy level middleware.

Inside the token analyzer, required metadata will be extracted from the (JavaScript Object Notation) JSON Web Tokens (JWT) and fed to a rule-based algorithm with code-level security metrics. The algorithm completely depends on security metrics and the token metadata such as token scope, audience, expiration. The algorithm's first step will store those security metadata into the array and sort those data with the help of security metrics provided by the code analyzer. In the second stage, the outputs that come on the token analyzer will provide input for the policy generator; inside the policy generator, it will maintain the Comma Separated Values (CSV) file to store the security metrics. With the help of those security metrics, it will generate proxy policies. The creation of policies varies from role base user level to access token level. At the base level, this component can generate a policy for the role-based user level. The control agent is exposed to the DevOps environment. It can apply the above-generated policies into the Kubernetes cluster via "*kubectl*".

C. *Codebase Analyzer*

The code analyzer component connects metrics to proxy-level middleware created as a hybrid of two primary components.

- A front-end web application for handling all the user interactions.
- A backend server that handles all the heavy work of code analyzing.

The main objective of this component is to provide necessary matrices and results for the Token Manager. This tool uses several ways to identify the level of scope that needs to be included in the access tokens for a specific microservice.

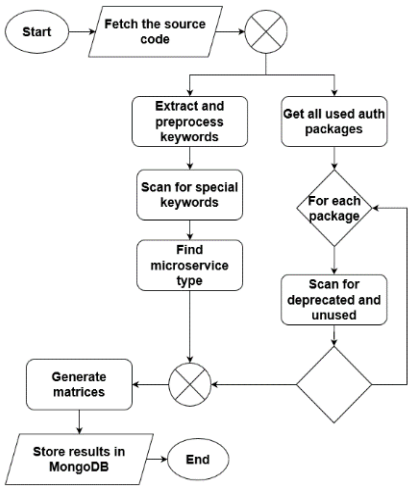

Fig. 3. Inner workflow of the Code Analyzer tool

This tool only supports JavaScript codebases and fetching code from GitHub and GitLab. The user can view all the inter processes through a terminal-like interface. This will log all the messages sent by the backend service and display them in a proper manner. Then the tool will fetch the source code and in this step, tool clone the GitHub/GitLab repository to a temporary location in the server and select the required directory using the user entered sub path. For the analyzing part, Code Analyzer does the following tasks.

- The tool looks for the most common and special keywords (payment gateway names, service endpoint names, different protocol names) used to identify distinct microservices and called endpoints.
- Scans all dependency types (normal, peer, optional, bundled) except development dependencies. Using the "package.json" file, the tool analyzes details about used packages and can identify them. The tool needs to analyze all the codebase to identify unused packages and deprecated packages, along with the version details, version gap and package details.

After generating required matrices for the Token Manager, the tool saves the results to a No-SQL database for future access to those other components as JSON types of objects. The reason to use a No-SQL database is that it is a proper way to handle complex JSON results.

D. Transaction Tracer with Root Cause Analysis

The main objective of the tracing and root cause analysis component is that if any failure occurs in the application, find those error sources quickly, efficiently and make it easy for developers to address them immediately. To accomplish the task of the tracing component uses application-level monitoring information and service-level communication data. These data are gathered using distributed tracing. The process of tracking a request as it goes through the system is known as tracing. When a request arrives at the ingress gateway [22], some meta-information like request-id, span id, and trace headers are added. It is subsequently included in any communication within the cluster initiated for the specific request. In the proposed approach, the proxies in each pod can then look at that meta information and collectively determine the request's route across the system. Tracing can give information on latencies, error rates, number of requests, and more using other information kept at the proxy, such as timestamps and status codes. It includes application-level metrics.

IV. RESULTS AND DISCUSSION

To evaluate the optimization of the above-proposed model, authors have used a sample microservice-based e-commerce system, including functionalities like customer, products, order processing, inventory, cart, and payment-gateway microservices. It is deployed in an Azure Kubernetes Service(AKS) consisting of three nodes (one master and two worker nodes) with standard B2s VM size [23]. The application is built with JavaScript Ecma Script 7 (ES7) and Java Springboot frameworks. Each service made use of remote databases. The communication mechanism of the system is REST APIs and JSON format. All services are wrapped with Docker containers. To execute the above-mentioned e-commerce application, the authors developed a load generator responsible for distributing an equal load to each microservice. Behavior Driven Development (BDD) test cases were added to define random user context. Therefore BDD test cases can generate erroneous requests. In order to evaluate the proposed security optimization model, the CPU and memory usages of the pods were compared with the existing auth method over newly discovered auth methodologies depicted in Table III. Moreover, request-response latency duration has been assessed per each auth method.

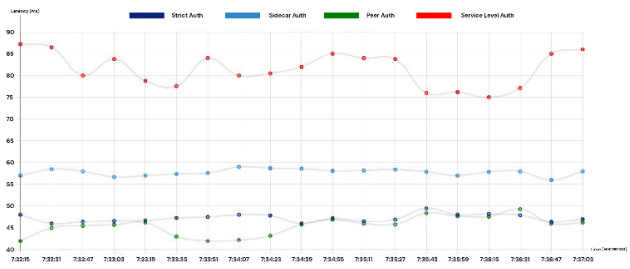

Fig. 4. Request – Response latencies of new auth methods and existing auth methods

Time-series data were used to calculate the performance evaluation matrices discussed above. The integrated Prometheus server within the same Kubernetes cluster collects those time-series data analytics from the pods. According to the findings presented in Fig. 4, it is evident that newly installed auth strategies (Strict Auth, Peer Auth, and Sidecar Auth) shorten the time it takes for an erroneous request to be processed and responded to.The request-response latency for Sidecar Auth, Strict Auth, and Peer Auth is between 35 and 50 milliseconds, while Sidecar Auth is taking around 55-60 rage of milliseconds. There is a significant impact on Existing Auth, which is experiencing a delay of 75-95 range of milliseconds. Above all, time-series data is taken each and every second and presents an average delay for the previous 05 minutes from the current time. Table III presents an abstract summary of the above-obtained results. The proposed model secured arround 30% upto 45% of latency duration rather than the existing approach across above 03 new auth methodloies.

TABLE IV. OPTIMIZED PRECENTAGES OF THE LATANCY

Auth Type	AGV request response duration	Optimized portion than the Existing Auth
Strcit Auth	47.258	42.14 %
Peer Auth	45.5	44.17 %
Sidecar Auth	57.789	29.09 %

However, the latency that occurred due to an erroneous request is never entirely dependent on the duration of the response. CPU consumption and memory usage are the Kubernetes pod's resource utilization matrices, which could significantly impact the request-response duration. Table V and Table VI depict the CPU usage and the memory usage of the Kubernetes pod, respectively. The minimum and maximum measurements are taken within 10 seconds from time-series data to calculate the relative resource utilization. Relative to the existing Auth mechanism, Sidecar Auth consumes a higher consumption rate of resources, while Peer Auth and Strict Auth consume fewer. CPU consumption is represented in CPU seconds and MegaBytes (MB) for memory usage.

TABLE V. RELATIVE PRECETAGES OF CPU OPTIMIZATION

Auth Type	MIN CPU Usage	MAX CPU Usage	AVG CPU Usage	Relative CPU Optimization
Sidecar Auth	0.0164	0.0182	0.0173	+ 50.72 %
Exsisting Auth	0.0063	0.0074	0.0069	0
Peer Auth	0.0022	0.0032	0.0027	- 43.75 %
Strict Auth	0.0032	0.0043	0.0038	- 44.92 %

TABLE VI. RELATIVE PRECENTAGES OF MEMORY OPTIMIZATION

Auth Type	Min Memory Usage	Max Memory Usage	Average Memory Usage	Relative Memory Optimization
Sidecar Auth	0.0033	0.0046	0.0039	+ 16.66 %
Existing Auth	0.0015	0.0021	0.0018	0
Peer Auth	0.0005	0.0011	0.0008	- 55.55 %
Strict Auth	0.0007	0.0012	0.0009	- 50.00 %

The reason for the increased CPU and memory use in the Sidecar Auth is that a sidecar is an extra container operating within the Kubernetes pod. Hence thereby needing additional computing power and storage. In terms of the request-response latency represented in Fig. 4, even though Sidecar Auth is somewhat quicker than Existing Auth, it results in inefficient resource utilization.


```
{
    "auth_patterns": {
        "total_auth_percentage": 16.97,
        "auth_patterns_per_package": [
            {
                "name": "passport",
                "letter_percentage": 9.07,
                "package_patterns_percentage": 64.29,
                "total_pattern_count": 9
            },
            {
                "name": "jsonwebtoken",
                "letter_percentage": 6.25,
                "package_patterns_percentage": 35.71,
                "total_pattern_count": 8
            },
            {
                "name": "jwt-decode",
                "letter_percentage": 1.65,
                "package_patterns_percentage": 21.43,
                "total_pattern_count": 3
            }
        ]
    }
}
```


Fig. 5. Sample generated result of Codebase Analyzer

Fig. 5 illustrates a sample JSON output from Codebase Analyzer. This output shows the percentage of auth implementations in each of the auth packages used in the codebase. The term "Letter Percentage" refers to the percentage of letters used in the codebase as a percentage of the total source code. Due to the fact that we used Regular Expressions (RegEx) patterns to identify auth implementations, "Package Patterns Percentage" refers to the percentage of RegEx patterns that were matched, and "Total Pattern Count" refers to the total number of matched patterns. Also, this tool gives detailed security measurements of the packages used in the source code (Due to the size of the output, Fig. 5 only shows auth implementation measures).

Those newly developed authentication strategies discussed previously have significant quantitative value. While the outputs generated by the Codebase Analyzer have quantitative value, when we consider those auth implementation percentages when determining the most appropriate auth strategy, developers must consider them qualitatively. That is, if the "Total Auth Percentage" is too high, developers should consider using authentication methods that generally have low average CPU usage, and vice versa. As an overall, since authentication and authorization process are haddling outside from service's business logic, the newly proposed model ensures the optimization of inter service commucation among

the pods. As a result of the research findings, the sidecar strategy is only acceptable for a limited number of use cases due to its high CPU and memory consumption. is well-suited for services that prioritize security, such as financial transaction services and payment gateways. Because these services require a greater emphasis on security characteristics than on performance and are not often scaled horizontally, if the horizontal scaling rate is increased, it may have an adverse effect on the CPU and memory of the Kubernetes cluster's worker nodes. For performance-critical microservices like user management and product management services often scaled horizontally, either Strict Auth or Peer Auth may be the best match.

V. CONCLUSION

This research focuses on a cloud-native solution to reduce network latency caused by erroneous requests from microservices orchestrated by the Istio service mesh. While the code analyzer component assures software security, the other components focus on network security. The authors present the core architectural concepts and methodologies used in this regard, and results proceed through the proposed model on the cloud-hosted Kubernetes cluster. According to the results obtained from CPU usage, memory usages, request-response latency duration, and usage level of auth dependencies from the business services, the proposed model has cleared a solution of optimizing the security matrices with a positive impact on its deployed architecture. The paper's outcome address the impact of adding a WASM-enabled envoy extension to the Istio service proxy to minimize the latency associated with auth configurations. However, further study is necessary to evaluate and validate the effectiveness of the practical experimental results. The reverse proxy could be enhanced by distributing the proxy cache across multiple layers. After enabling the tracing option in the cluster, a small amount of performance latency will be generated. However, there are publications currently available to assist in reducing trace latency

VI. REFERENCES

[1] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," in IEEE Software, vol. 35, no. 3, pp. 24-35, May/June 2018, doi: 10.1109/MS.2018.2141039.

[2] Brent Smith, Greg Linden, “Two Decades of Recommender Systems at Amazon.com”, in IEEE, vol. 21, pp. 12-18, May-June-2017, doi: 10.1109/MIC.2017.72

[3] Mandal, G. K., Diroma, F., & Jain, R. (2017). Netflix: An In-Depth Study of their Proactive & Adaptive Strategies to Drive Growth and Deal with Issues of Net-Neutrality & Digital Equity. *IRA-International Journal of Management & Social Sciences (ISSN 2455-2267)*, *8*(2), 152. https://doi.org/10.21013/jmss.v8.n2.p3

[4] Ladislav Burita, “Information Analysis on Facebook”, vol. 1, Oct 2019, doi: 10.23919/KIT.2019.8883471

[5] Rishi Srinivas, B. Ankayarkanni, R. Sathya Bama Krishna, “Uber Related Data Analysis using Machine Learning”, May 2021, doi: 10.1109/ICICCS51141.2021.9432347

[6] D. Trihinas, A. Tryfonos, M. D. Dikaiakos and G. Pallis, "DevOps as a Service: Pushing the Boundaries of Microservice Adoption," in IEEE Internet Computing, vol. 22, no. 3, pp. 65-71, May./Jun. 2018, doi: 10.1109/MIC.2018.032501519.

[7] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," 2015 10th Computing Colombian Conference (10CCC), 2015, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476.

[8] Wu, A. (2018). Running Your Modern .NET Application on Kubernetes.

[9] A. Pereira Ferreira and R. Sinnott, "A Performance Evaluation of Containers Running on Managed Kubernetes Services," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2019, pp. 199-208, doi: 10.1109/CloudCom.2019.00038.

[10] Siriwardena, P., & Dias, N. (n.d.). *Microservices Security in Action*.

[11] Z. Houmani, D. Balouek-Thomert, E. Caron and M. Parashar, "Enhancing microservices architectures using data-driven service discovery and QoS guarantees," 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), 2020, pp. 290-299, doi: 10.1109/CCGrid49817.2020.00-64.

[12] An Analysis of the Transport Layer Security Protocol Thyla van der Merwe. (2018).

[13] Lodderstedt T., McGloin M., Hunt P.: OAuth 2.0 Threat Model and SecurityConsiderations.

[14] R. Melton, "Securing a Cloud-Native C2 Architecture Using SSO and JWT," 2021 IEEE Aerospace Conference (50100), 2021, pp. 1-8, doi: 10.1109/AERO50100.2021.9438218.

[15] “RBAC-based Access Control for SaaS Systems” in Proc 2nd International Conference on Information Engineering and Computer Science, TBD Wuhan China

[16] Balamurugan B et al, “Enhanced Framework for Verifying User Authorization and Data Correctness Using Token Management System in the cloud”, in proc: International Conference on Circuit, Power and Computing Technologies

[17] Sigelman, B. H., Barroso, A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). Google Technical Report dapper Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.

[18] Höglund, J. (n.d.). AN ANALYSIS OF A DISTRIBUTED TRACING SYSTEMS EFFECT ON PERFORMANCE Jaeger and OpenTracing API.

[19] Burns, B., & Google, D. O. (n.d.). Design patterns for container-based distributed systems.

[20] Dattatreya Nadig, N. (2019). Testing Resilience of Envoy Service Proxy with Microservices. In *DEGREE PROJECT IN TECHNOLOGY*.

[21] Extending Envoy with Wasm from start to finish Ed Snible / @esnible / IBM Research

[22] Mara Jösch, R. (n.d.). Managing Microservices with a Service Mesh An implementation of a service mesh with Kubernetes and Istio. In *DEGREE PROJECT COMPUTER SCIENCE AND ENGINEERING*.

[23] Azure Documentation - B-series burstable VM support in AKS now available | Azure Blog and Updates | Microsoft Azure

[24] “The Rise of Service Mesh Architecture”, dzone.com, 2019. [Online]. Available: https://dzone.com/articles/the-rise-of-service-mesh-architecture [Accessed: Aug. 26, 2021]

[25] Goran Piskachev, Lisa Nguyen Quang Do, Eric Bodden “Codebase-adaptive detection of security-relevant methods”, [online document],2019.Available: https://dl.acm.org/doi/abs/10.1145/3293882.3330556 [Accessed: Aug. 26, 202z