Automated Security Analysis for Microservice Architecture

Nacha Chondamrongkul*, Jing Sun† and Ian Warren‡
School of Computer Science, University of Auckland
Auckland, New Zealand
*ncho604@aucklanduni.ac.nz, †jing.sun@auckland.ac.nz, ‡i.warren@auckland.ac.nz

***Abstract*—Designing a software system that applied the microservice architecture style is a challenging task, as its characteristics are vulnerable to various security attacks. Software architect, therefore, needs to pinpoint the security flaws in the design before the implementation can proceed. This task is error-prone as it requires manual analysis on the design model, to identify security threats and trace possible attack scenarios. This paper presents an automated security analysis approach for microservice architecture. Our approach can automatically identify security threats according to a collection of formally defined security characteristics and provide an insightful result that demonstrates how the attack scenarios may happen. A collection of formally defined security characteristics can be extended to support other security characteristics not addressed in this paper.**

***Index Terms*—Microservice Architecture; Security Analysis; Ontology Web Language; Model Checking**

I. Introduction

Microservice architecture has been an emerging trend for both research and industry. Unlike traditional Service Oriented Architecture (SOA) that are usually developed and deployed as a single executable artefact, each service in microservice architecture can be independently developed, deployed and executed on its own environment. However, implementing the microservices has no formal guidance, so many security challenges are posed due to the following reasons. First, the microservice architecture embraces fine granularity. The complex system is therefore composed of a great number of services accessible across the network [1]. Second, the service is usually deployed on the cloud-based container so it must share the same kernel with other unknown containers. The system security hence relies heavily on how safe these containers are [2]. Third, as the system relies on invoking services, authentication and authorization control must be adequately in place [1]. Fourth, as the services are dispersed across the network, the data transferring over the network is vulnerable to disclosure and tampering without enough network security control [2]. Having the right architecture design and deployment configuration is crucial to secure microservices.

If we can identify security flaws in the microservice system at the early development stage, we would be able to identify and mitigate security flaws at the architecture design level before they are propagated to the implementation [3]. At the design phase, the security flaws can be identified by performing the architecture security analysis using software architecture design model [2]. Although there are approaches [4] [5] [6] proposed to support this analysis, they require analysis logic hardcoded in the implementation of tools. Almorsy et al. [7] have proposed an extensible analysis tool. However, these tools focus on the analysis of software architecture model in general. Microservice architecture is composed of a large number of services dispersing across the network. The results from existing tools are lack of insightful information that guides software engineer to trace through the design thoroughly. Some approaches [8] [9] [10] have been proposed to analyse the microservice architecture in particular, but they focus on checking specific behaviour according to the security policy. To our knowledge, at present, there exists no approach that can analyse security vulnerabilities in the microservice architecture based on an extensible set of security characteristics and yet provides an insightful result.

This paper presents an approach that combines ontology reasoning and model checking technique to support security analysis for microservice architecture. Our approach aims at automatically analysing security characteristics and gives an insightful result that demonstrates how the attacks happen.

The contributions of this paper can be summarised as follows. First, the formal modelling of software architecture design is proposed to describe structural and behavioural aspect of microservice architecture. Second, a set of the formal description of security characteristics are presented and used to identify security vulnerabilities, metrics or controls. This set is extensible to support other security characteristics not addressed in this work. Third, the analysis tool has been developed to support modelling the microservice architecture design and analysing security.

The rest of this paper is organized as the following sections. Section II presents details of our approach. Section III presents the illustration of our approach with an example system. This paper is concluded in Section IV.

II. Formal Security Analysis

The overall process of our approach is shown in Figure 1. Firstly, the modeller tool is used to create a model of software architecture design. Secondly, the modeller generates the structural model in Ontology Web Language (OWL) and the behavioural model in Architecture Description Language (ADL). Thirdly, the ontology reasoner performs reasoning on

978-1-7281-4659-1/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSA-C50368.2020.00024

the structural model and identifies security characteristics in the design elements such as component and connector. This reasoning is based on the the formal definition kept in the Arch repository. These security characteristics are predefined as the ontology classes and rules that capture specific structural features of the design elements. Fourthly, the liveness properties are generated to prove the security scenarios. Fifthly, the generated properties are verified by the model checker against the behavioural model. Lastly, the result is given to demonstrate how the system responses to the attack.

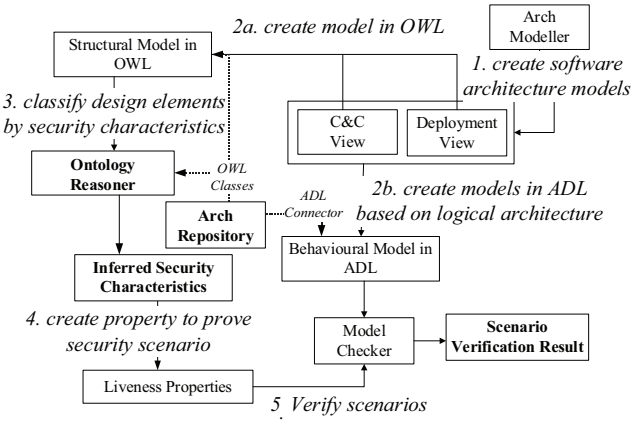

Fig. 1. Overall process of our security analysis

A. Microservice Architecture Modelling

This modelling aims at describing the structures and behaviours in the architecture design to support detecting security characteristics and generating security scenarios. This approach applies the ontology representation to illustrate the structure in architecture design as presented in [11] because of its expressiveness in defining the structure and its scalability to a large volume of data. It is therefore suitable to manifest the microservice architecture design that has a large number of components. The ontology representation in OWL is used to describe the structural model that consists of two architecture views, namely, Component & Connector (C&C) view and deployment view.

Figure 2 depicts the ontology classes in the repository that supports both views in structural modelling. In C&C view, component represents the computation unit that provides one or more services and connector represents the connectivity that links the components to each other. The component has one or more ports representing its interface, while the connector has one or more roles representing parties involved in the interaction. Port is to be attached to one or more roles, to link a component to another component. This structure supports mapping to the behavioural model in Architecture Description Language (ADL). The deployment view describes how components are deployed to the infrastructure; it therefore includes physical elements such as device, execution environment and communication link. The device representing the physical node consists of one or more execution environments, where

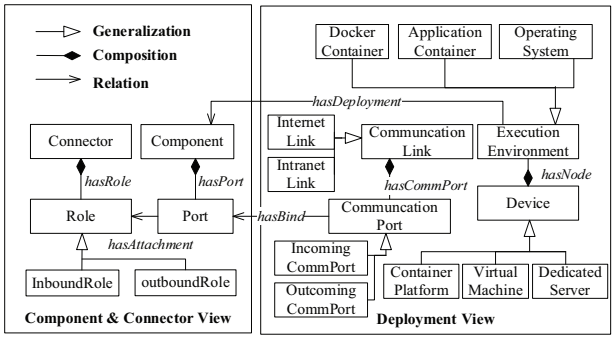

Fig. 2. Ontology Classes for Structural Modelling

the components are deployed and executed. For example in the microservice architecture a component with a service may be deployed on an execution environment like Docker container. The communication link is the physical network link that allows the components to communicate with each other; it has communication ports representing the network ports.

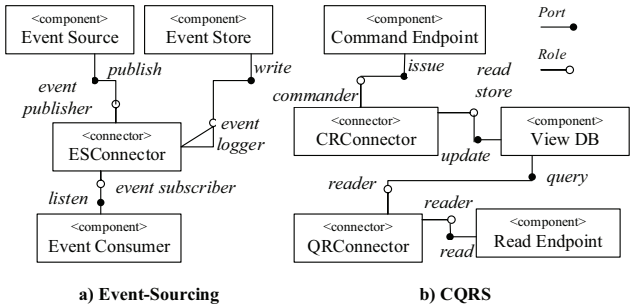

Fig. 3. Patterns used in Microservice Architecture

To support the modelling of microservice architecture, two architecture patterns, namely Event-Sourcing and Command and Query Responsibility Segregation (CQRS) [12], are formally defined[1]. As shown in Figure 3, specific types of component, connector, port, role and relationship are formally defined as the ontology classes for these patterns (extended from the hierarchy in Figure 2).

$$eventsource \equiv Component \sqcap \exists\, hasPort\ publish$$
$$eventstore \equiv Component \sqcap \exists\, hasPort\ write$$
$$eventconsumer \equiv Component \sqcap \exists\, hasPort\ listen$$
$$ESConnector \equiv Connector \sqcap \exists\, hasRole\ eventpublisher \sqcap \exists\, hasRole\ eventlogger \sqcap \exists\, hasRole\ eventsubscriber$$
$$eventpublisher \sqsubseteq OutboundRole$$
$$eventlogger, eventsubscriber \sqsubseteq InboundRole$$

Due to the page limit, we can only present the formal definition of ontology classes defined for the event-sourcing pattern as shown above in description logic. For event-sourcing pattern, the *ESConnector* represents the connectivities among three component types. The *Event Source* component is where the event is originated. After the event is published, the

[1]Complete patterns in OWL and ADL for microservice architecture can be found at http://bit.ly/34gH2dq

event is recorded on the *Event Store* component and the *Event Consumer* component is notified. Roles are defined with *OutboundRole* representing the role that sends message out or *InboundRole* representing the role that receives message. CQRS is also formally defined similarly. The structural model of a software system can be defined by creating ontology individuals according to the ontology classes in Figure 2.

The behavioural model is defined in ADL according to the structure in C&C view. We have used Wright# [13] as an ADL and PAT model checker. In the repository, the behaviour of connector type is defined in Wright# as Communicating Sequential Process (CSP). Each role is defined with a CSP process. The channel input (?) and output (!) are used to describe the interaction between roles. The connector types of event-sourcing can be defined in Wright#, as shown below. The connector types for CQRS are also similarly defined.

connector *ESConnector* {
role *eventpublisher*(*j*) = *process* → *pevt*!*j* → *sevt*?*j* → *bevt*!*j* → *broadcast* → *Skip*;
role *eventsubscriber*() = *bevt*?*j* → *process* → *eventsubscriber*();
role *eventlogger*() = *pevt*?*j* → *process* → *sevt*!*j* → *persist* → *Skip*; }

The behavioural model of a microservice system can be defined in ADL to describe the behaviour of interaction. With the defined connector types shown above, we can define the behavioural model as follows: 1) The components participating in the system is defined with the behaviour of their ports, 2) The system configuration is defined to describe the interactions by connecting the connector's role with the component's ports. More details of behaviour modelling can be found in [13].

B. Security Characteristics Detection

Security characteristics specific to microservice architecture are selected to be identified in the design. The first four characteristics represent security metrics that measure how secure the system is, while the last two characteristics represent security vulnerabilities that are used to analyse the attack scenarios. The definition of security characteristics follows the guidelines found in the literature [3] [7] [2]. The security characteristics are semantically defined as ontology classes in the repository, as shown in the subsections below. With the ontology classes, the reasoner can identify ontology individuals representing component or connectors that are semantically inferred. These set of ontology classes are extensible by defining a new class that inherits existing classes or conditionally matches some properties through ontology rule.

1) Attack Surface: is the component that opens to the public network or is deployed on the container hosted on the public cloud; as the component is vulnerable to malicious containers deployed on the same host [2].

$$\begin{aligned} \mathit{AttackSurface} &\equiv \mathit{Component} \sqcap \exists\, \mathit{hasPort}\ (\mathit{Port} \sqcap \exists\, \mathit{isBindTo} \\ &\quad (\mathit{IncomingCommPort} \sqcap \exists\, \mathit{isCommPortOf}\ \mathit{InternetLink})) \\ &\equiv \mathit{Component} \sqcap \exists\, \mathit{isDeployedOn} \\ &\quad (\mathit{DockerContainer} \sqcap \exists\, \mathit{isNodeOf} \\ &\quad (\mathit{ContainerPlatform} \sqcap \mathit{PublicDevice})) \end{aligned}$$

2) Defence In Depth: is assessed by checking whether the critical components have security controls applied [7].

$$\begin{aligned} \mathit{DefenceInDepth} &\equiv \mathit{CriticalComponent} \sqcap \exists\, \mathit{hasPort}\ (\mathit{Port} \sqcap \\ &\quad \exists\, \mathit{isBindTo}(\mathit{IncomingCommPort} \sqcap \mathit{SecureCommPort})) \end{aligned}$$

3) Least Privilege: is measured by the number of components that can access critical components [7]. An ontology rule in Semantic Web Rule Language (SWRL) is defined to select these components.

$$\begin{aligned} &\mathit{hasPort}(\mathit{comp}1, p1) \wedge \mathit{hasPort}(\mathit{comp}2, p2) \wedge \mathit{hasAttachment}(p1, r1) \\ &\wedge \mathit{hasAttachment}(p2, r2) \wedge \mathit{hasRole}(\mathit{con}, r1) \\ &\wedge \mathit{hasRole}(\mathit{con}, r2) \wedge \mathit{OutboundRole}(r1) \wedge \mathit{InboundRole}(r2) \\ &\wedge \mathit{CriticalComponent}(\mathit{comp}2) \rightarrow \mathit{LeastPriviledge}(\mathit{comp}1) \end{aligned}$$

4) Compartmentalization: is measured by the number of components that performs authentication and authorization when its service is called [7].

$$\begin{aligned} \mathit{Compartmentalization} &\equiv \mathit{Component} \sqcap \exists\, \mathit{hasPort}\ (\mathit{Port} \sqcap \exists\, \mathit{isBindTo} \\ &\quad (\mathit{AuthenticatedCommPort} \sqcap \mathit{AuthorizedCommPort})) \end{aligned}$$

5) Denial of Service (DOS): is a security attack that makes a system or its services and resources unavailable for legitimate users. We can check this by finding the component that is an attack surface and has no input sanitization [3].

$$\begin{aligned} \mathit{DenialOfServiceConnector} &\equiv \mathit{Connector} \sqcap \exists\, \mathit{hasRole} \\ &\quad (\mathit{InboundRole} \sqcap \exists\, \mathit{isAttachmentOf}(\mathit{Port} \sqcap \exists\, \mathit{isPortOf} \\ &\quad \mathit{AttackSurface} \sqcap \exists\, \mathit{isBindTo}\ \mathit{NoInputSanitizedCommPort})) \end{aligned}$$

6) Man in the Middle (MITM): is when attackers intercept the communication between components. The connector is vulnerable to MITM when it transfers data as plain text over the public network [2].

$$\begin{aligned} \mathit{ManInMiddleConnector} &\equiv \mathit{Connector} \sqcap (\exists\, \mathit{hasLinkVia} \\ &\quad (\mathit{PlainLink} \sqcap \mathit{InternetLink})) \end{aligned}$$

C. Attack Scenarios Generation

In our approach, the attack scenarios can be generated and tracked by executing the model checker on the behavioural model in ADL asserted with liveness properties. The liveness property is generally used to check whether the desirable states eventually happen. In our approach, the liveness property is defined to trace the attack scenarios such as DOS and MITM. The liveness property can be defined in Linear Temporal Logic (LTL) formula, as shown below.

$$\Box(\mathit{adversary.vconn.outrole.vevnt} \rightarrow \Diamond\ \mathit{targetcomp.inport.cevnt})$$

where $\Box$ (always) and $\Diamond$ (eventually) is an LTL operator; *adversary* represents an adversary component inserted to the model and *targetcomp* represents a target component connected to a vulnerable connector *vconn* (identified by the ontology reasoner); *outrole* represents a role of *vconn* attached to the *adversary*; *inport* is a port of *targetcomp*; *vevnt* represents an event triggered by the *adversary* and *cevnt* represents an event triggered by *targetcomp*. In other words, this property checks whether a *targetcomp* can be eventually called by the *adversary* through *vconn* connector. The negation of this property can be processed by the model checker to retrieve a state trace showing how the system responses to the attack. The use of model checking technique helps us to find other components that cannot be identified by the ontology reasoner but may have consequent impact from the attack.

III. Illustration

We have developed features in ArchModeller[2] tool to facilitate our analysis process. These features support modelling the architectural design in the graphical diagrams and consolidating the analysis results. Using this tool, we created the design diagrams for Sock Shop[3], a reference system of the microservice architecture. Two different deployment configurations are created to compare. After the models have been processed by the ontology reasoner, the numbers of found security characteristics are given and can be used to render a radar chart shown in Figure 4. ▲ means that the higher the value is, the more secure the deployment configuration is. While ▼ means that the less the value is, the more secure the deployment configuration is. This chart assists the trade-off analysis by comparing different deployment configurations[4]. In this example, one may decide to select the deployment configuration 1 as it is more secure.

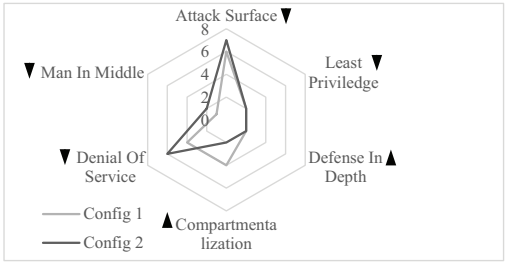

Fig. 4. Security Radar Chart

The result of ontology reasoning helps us to identify a connector vulnerable to DOS attack. This connector (*orderwire*) supports ordering product, so a liveness property to prove this scenario is defined as $\Box(AdversaryDOS1.orderwire.requester.request \rightarrow \Diamond OrderService.postorder.check)$, where *AdversaryDOS1* represents the adversary's component, and *OrderService* represents a service processing customer order. The negation of this property gives a state trace as shown below. This state trace starts with the *request* event triggered by attached *requester* role to consume a service on *OrdersService* that consequently invoked *OrdersLog*, *OrdersCommand* and *OrdersDB* respectively. It can be seen from the state trace that not only *OrdersService* is affected by this DOS attack but also other components that have consequent impact.

$init \rightarrow AdversaryDOS1.orderwire.requester.request$
$\rightarrow orderwire.req!64 \rightarrow orderwire.req?64$
$\rightarrow OrdersService.orderwire.responder.invoke \ldots$
$\rightarrow OrdersLog.logorder.orderlogged \ldots$
$\rightarrow OrdersCommand.insertorder.saveorder \ldots$
$\rightarrow OrdersDB.writeorder.orderwritten \ldots$

A preliminary evaluation has been conducted on different models to assess the accuracy of our approach[5]. We have found that the precision of detection relies on how accurate the security characteristics are defined. A more comprehensive evaluation is yet required to understand its performance better.

[2]Arch Modeller can be found at http://bit.ly/2m3LITT

[3]Sock Shop can be found at https://microservices-demo.github.io/

[4]The models representing two deployment configuration of Sock Shop can be found at http://bit.ly/2LOntmA

[5]The evaluation results can be found at http://bit.ly/2vLmuyd

IV. Conclusion

We introduce a security analysis approach for microservice architecture design. Our approach can automatically identify security threats and provides an insightful result that helps to analyse possible impacts. With the support of the analysis tool, the architecture design of the microservices can be modelled and analysed seamlessly. Other security characteristics not addressed in this paper can be analysed by extending our set of the ontology definition of security characteristics, without modifying and rebuilding the source code of the tool. The formally defined security characteristics in the repository can be expanded by security engineers or security communities. This repository could provide software engineers with a standard way to automatically analyse and find security flaws in microservice architecture design.

References

[1] T. Yarygina and A. H. Bagge, "Overcoming security challenges in microservice architectures," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, March 2018, pp. 11–20.

[2] D. Yu, Y. Jin, Y. Zhang, and X. Zheng, "A survey on security issues in services communication of microservices-enabled fog applications," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 22, p. e4436, 2019, e4436 cpe.4436.

[3] P. Nkomo and M. Coetzee, "Software development activities for secure microservices," in *Computational Science and Its Applications – ICCSA 2019*, S. Misra, O. Gervasi, B. Murgante, E. Stankova, V. Korkhov, C. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, and E. Tarantino, Eds. Cham: Springer International Publishing, 2019, pp. 573–585.

[4] J. Gennari and D. Garlan, "Measuring attack surface in software architecture (cmu-isr-11-121)," 2012.

[5] L. Grunske and D. Joyce, "Quantitative risk-based security prediction for component-based systems with explicitly modeled attack profiles," *Journal of Systems and Software*, vol. 81, no. 8, pp. 1327 – 1345, 2008.

[6] M. Bunke and K. Sohr, "An architecture-centric approach to detecting security patterns in software," in *Engineering Secure Software and Systems*, Ú. Erlingsson, R. Wieringa, and N. Zannone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 156–166.

[7] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 662–671.

[8] K. A. Torkura, M. I. Sukmana, and C. Meinel, "Integrating continuous security assessments in microservices and cloud native applications," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: ACM, 2017, pp. 171–180.

[9] T. Asik and Y. E. Selcuk, "Policy enforcement upon software based on microservice architecture," in *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, June 2017, pp. 283–287.

[10] C. Gerking and D. Schubert, "Component-based refinement and verification of information-flow security policies for cyber-physical microservice architectures," in *2019 IEEE International Conference on Software Architecture (ICSA)*, March 2019, pp. 61–70.

[11] N. Chondamrongkul, J. Sun, and I. Warren, "Ontology-based software architectural pattern recognition and reasoning," in *30th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, June 2018, pp. 25–34.

[12] S. Millett and N. Tune, *Patterns, Principles, and Practices of Domain-Driven Design*. Wiley, 2015.

[13] N. Chondamrongkul, J. Sun, and I. Warren, "Pat approach to architecture behavioural verification," in *31th International Conference on Software Engineering and Knowledge Engineering*, July 2019, pp. 187–192.