Towards Model-driven Resolution of Security Smells in Microservices

Philip Wizenty[1] · Francisco Ponce[2,3] · Florian Rademacher[4] · Jacopo Soldani[5] · Hernán Astudillo[2,3] · Antonio Brogi[5] · Sabine Sachweh[1]

[1] IDiAL Institute, University of Applied Sciences and Arts Dortmund, Germany
[2] Universidad Técnica Federico Santa María, Valparaíso, Chile
[3] ITiSB, Universidad Andrés Bello, Viña del Mar, Chile
[4] Software Engineering, RWTH Aachen University, Aachen, Germany
[5] University of Pisa, Pisa, Italy

✉ philip.wizenty@fh-dortmund.de

1 Introduction

Microservice Architecture (MSA) [7] is becoming popular among companies like Amazon, Netflix, and Twitter. It allows full utilization of cloud computing and alignment with DevOps practices [1]. MSA is an architectural style for distributed, dynamic, and fault-resilient software systems. It also brings new security challenges, including the identification of *security smells* [9].

The presence of security smells can be a sign of bad design choices (often made unintentionally), which may compromise the security of the entire software system [9]. However, these security smells can be fixed by implementing established refactoring techniques that enhance security without disrupting client functionality. While addressing security smells can be challenging, it can lead to improved software system quality [3].

Detecting security smells in microservices, and reasoning on how to refactor them is time-consuming and error-prone. This follows from the inherent complexity of MSAs, which involves tens to hundreds of services, which interact with one another. For this reason, we here propose to use Model-Driven Engineering (MDE) [4] to automatically detect and refactor two of the most common smells for microservice security [9]. Our proposal is based on extending the Language Ecosystem for Modeling Microservice Architecture (LEMMA) [11], an ecosystem for microservices design, development, and operation with the functionalities to detect those security smells. More precisely, we present a method for dealing with security smells in microservices using MDE, which is based on extending LEMMA to incorporate security aspects of microservices and on developing an automated process for analyzing LEMMA models to identify the two common security smells and suggest refactorings to fix them.

The remainder of the paper is structured as follows. Section 2 provides information about security smells and LEMMA in general. The following Section 3 introduces our approach for modeling, detecting, and resolving security smells in LEMMA models. Sections 4 and 5 present related work and draw some concluding remarks. Furthermore, this submission is based on our previous work [14], where interested readers can also find a feasibility assessment of the described approach.

2 Background

Security Smells for Microservices. An important task in software engineering is to be aware of security smells as they can be an inadvertent result of poor decision-making that could potentially compromise the overall

security of the software systems. In accordance with [9], we will discuss two commonly recognized security smells for microservices, *Publicly Accessible Microservices* and *Insufficient Access Control*. Additionally, we will introduce the appropriate refactorings to address these security concerns.

Publicly Accessible Microservices. This smell occurs when external clients have direct access to the microservices [8]. To prevent this, microservices should require authentication, asking users to provide their full credentials. Without this, there is an increased risk of confidentiality violations, lower maintainability, and reduced usability.

The integration of an API Gateway to reduce the attack surface and simplify authentication auditing is the suggested refactoring strategy to resolve this security smell. One such API gateway would then constitute the main access point to microservices APIs and enhance security by blocking external requests.

Insufficient Access Control. This smell occurs when microservices do not implement proper access control, which can lead to breaches of confidentiality [8]. Attackers can exploit this weakness to gain access to data/-functions they should not be able to access.

The suggested refactoring for this security smell is to use the OAuth 2.0 protocol to ensure proper access control for microservices. OAuth 2.0 implements a token-based authorization that allows the resource owner to authorize client access and verify permissions at each stage, automatically accepting or rejecting requests.

Language Ecosystem for Modeling Microservice Architecture. To cope with the challenges of MSA in MDE, LEMMA was conceived [10]. LEMMA offers a set of modeling languages that help address different MSA engineering concerns in models. Integrating them using an import mechanism allows for referencing between elements of heterogeneous models. It makes it easier to reuse them and increase the information content of captured *viewpoints* [10]. Our approach to resolving security smells in microservice architectures relies on the use of LEMMA modeling languages.

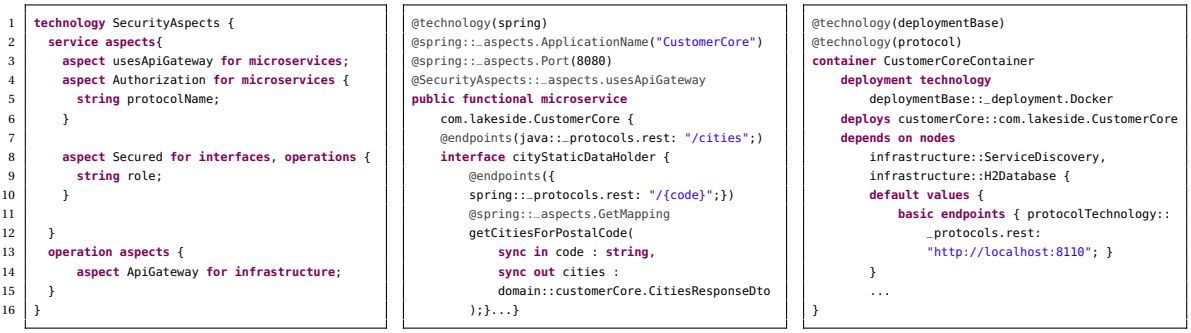
Technology Modeling Language (TML). LEMMA's TML targets the Technology Viewpoint on microservice architectures. That is, it allows for the construction of *technology models* that capture technology decisions related to microservices and their deployment, e.g., communication protocols and deployment technologies. Additionally, the TML supports the definition of *technology aspects* that apply to specific elements in LEMMA models, e.g., modeled microservices and their interfaces or infrastructure nodes.

Service Modeling Language (SML). LEMMA's SML reifies the Service Viewpoint in MSA engineering. It provides modeling concepts to specify microservices and their interfaces in *service models*. Among others, the SML integrates with the TML so that LEMMA service models can import LEMMA technology models to specify, e.g., protocol-dependent communication endpoints.

Operation Modeling Language (OML). LEMMA's OML focuses on MSA's Operation Viewpoint. The language supports the specification and configuration of microservice containers and infrastructure nodes, e.g., for service discovery or load balancing, in *operation models*. The OML integrates with the TML to cope with MSA's technology heterogeneity w.r.t. microservice operation and deployment [5].

3 Model-Driven Security Smell Resolution

Section 3.1 provides insights into the aspect-oriented modeling of security smells using LEMMA's TML. Section 3.2 elaborates the detection possibilities of security smells in MSA. Finally, Section 3.3 describes the user-guided refactoring process with LEMMA.



```
technology SecurityAspects {
  service aspects{
    aspect usesApiGateway for microservices;
    aspect Authorization for microservices {
      string protocolName;
    }

    aspect Secured for interfaces, operations {
      string role;
    }

  }
  operation aspects {
      aspect ApiGateway for infrastructure;
  }
}
```

(a)

```
@technology(spring)
@spring::_aspects.ApplicationName("CustomerCore")
@spring::_aspects.Port(8080)
@SecurityAspects::_aspects.usesApiGateway
public functional microservice
    com.lakeside.CustomerCore {
    @endpoints(java::_protocols.rest: "/cities";)
    interface cityStaticDataHolder {
        @endpoints({
        spring::_protocols.rest: "/{code}";})
        @spring::_aspects.GetMapping
        getCitiesForPostalCode(
            sync in code : string,
            sync out cities :
            domain::customerCore.CitiesResponseDto
        );}...}
```

(b)

```
@technology(deploymentBase)
@technology(protocol)
container CustomerCoreContainer
    deployment technology
        deploymentBase::_deployment.Docker
    deploys customerCore::com.lakeside.CustomerCore
    depends on nodes
        infrastructure::ServiceDiscovery,
        infrastructure::H2Database {
        default values {
            basic endpoints { protocolTechnology::
                _protocols.rest:
                "http://localhost:8110"; }
        }
        ...
}
```

(c)

Listing 1: LEMMAs technology (a), service (b) and operation (c) model for resolving security smells in MSA.

3.1 Modeling Microservice Security Aspects

The modeling of security configurations leverages LEMMA's TML *aspect* concept to incorporate metadata into a LEMMA service- or operation model. To start the modeling process in LEMMA, the modeler needs to select a security smell, e.g., *Publicly Accessible Microservices*. The next step is to analyze the smell and identify the underlying security configuration. For instance, the root cause of the mentioned security smell is the decision not to use an API Gateway [13] to expose the microservices interfaces.

The subsequent activity employs the obtained security configuration to model the security smell using LEMMA's TML. To proceed, one must choose a suitable technology model or create a new one that accommodates the derived configuration. The next step is to extend LEMMA's technology model by including all the derived security configurations. This extended model can be utilized to conduct automated tests that detect security smells in LEMMA models. The model presented in Listing 1(a) includes the modeled security configuration derived from the security smells.

The model in Listing 1(a) defines the `SecurityAspects` LEMMA technology model. The model contains the definition of `service aspects` assigned to the microservice concept of LEMMA's SML. More precisely, the `usesApiGateway` aspect explicitly defines that the microservices expose their interfaces via an API Gateway, as exemplified in Listing 1(b). Furthermore, to model an API Gateway with LEMMA's OML, the model specifies an `operation aspect` named `ApiGateway` to identify an infrastructure node in LEMMA's operation models as the corresponding component.

3.2 Detecting Security Smells

This section uses the created technology models (c.f. Listing 1(a)) as an indicator to detect security smells in LEMMA's service or operation models. In order to detect security smells in LEMMA models, it is necessary to model the microservices' API and deployment specification using the SML and OML (c.f. Listing 1(b,c)).

For example, Listing 1(b) depicts a concrete LEMMA service model for the `public functional microservice CustomerCore`. The microservice originates from the Lakeside Mutual[1] software system. The modeled microservice contains the `cityStaticDataHolder` interface as a specified single endpoint. The specification begins with an endpoint-specific extension of the interfaces URI for the `rest` over HTTP protocol. Additionally, the definition of the interface includes a Spring framework-specific aspect to the endpoint, indicating that the endpoint supports access via the HTTP-method *get*. Followed by the specification of the incoming and outgoing parameters, including their corresponding data types.

[1] `https://github.com/Microservice-API-Patterns/LakesideMutual`

The operation model in Listing 1(c) contains the deployment and operation specifications for the CustomerCore microservice with dependencies to infrastructure components, e.g., databases or service discoveries. The model begins with assigning the imported technologies to the `CustomerCoreContainer` to enable their usage in the deployment specification for the container. The container in LEMMA's OML is a component that encapsulates all relevant information for deploying a specific microservice via Docker deployment technology.

Runtime dependencies to infrastructural components, such as databases or service discoveries, are needed for scalability and synchronous or asynchronous service communication via different protocols. This includes dependencies to LEMMA infrastructure models, describing the deployment of a *Eureka*[2] service discovery and a *H2*[3] database. The last part of the listing defines `default values`, which are used for service operation. In this case, the `basic endpoint` for communication via HTTP and, therefore, as the beginning of the URI extended by endpoint specification of the service model.

After modeling the software systems architecture using LEMMAs SML and OML, they can be used to identify security smells. For this purpose, the service and operation models are analyzed via model-specific validators. The model validators for security smells can be explicitly enabled in the Eclipse-based model editor of LEMMA. According to the case that the validators detect a security smell in the models, they display a warning in the model editor stating the name of the security smell and possible strategies for resolving it.

3.3 Resolving Security Smells

This subsection introduces the activities to resolve the detected security smells in microservices. It also presents the user-guided process using LEMMA's modeling environment. The security smell resolving process starts after selecting the service or operation model containing the security smell. For each security smell, there are different strategies for resolution [9]. For instance, the *Publicly Accessible Microservices* security smell can be resolved by using an API Gateway or by disabling the public exposure of the service. It should be noted that exposing the microservice publicly can also be an intentional design decision that should be marked as intentional. As part of the resolution process, deliberately flagging a security smell as `ignored` is used to notify the user of its presence and acknowledge their design decisions.

Following the selection of the resolution strategy, the next activity is to preview the refactoring results specific to the chosen strategy. LEMMA's modeling editor provides a workflow to guide the user through this process. Finally, the last activity involves confirming the model changes and applying the resolution strategy to the involved models.

To illustrate the results of resolving the *Publicly Accessible Microservices* security smell using the strategy of including an API Gateway for public microservice exposure, an excerpt of the refactored operation model of the *Customer Core* service is shown in Listing 2(a). Generally, the operation model remains unchanged except for adding a `depends on` dependency to the infrastructural component of an `API Gateway`. Due to the inclusion of the API Gateway in the resolution strategy, a corresponding operation model is also created in the refactoring process of the *Customer Core* model for API Gateway integration.

Listing 2(b) specifies the deployment of the API Gateway infrastructure component using Netflix Zuul as a concrete technology. The model specifies dependencies to infrastructure nodes, e.g., a `Service Discovery` and the `Customer Core` microservice. The remaining content of the model defines `default values`, e.g., the hostname or operation port. That is used to define the access to the `API Gateway`.

[2] `https://github.com/Netflix/eureka`
[3] `https://www.h2database.com`

```
import ...

@technology(deploymentBase)
@technology(protocolTechnology)
container CustomerCoreContainer ...
    depends on nodes
        infrastructure::ServiceDiscovery,
        infrastructure::H2Database,
        infrastructure::APIGateway
    ...
}
```

(a)

```
import ...
@technology(Zuul)
APIGateway is Zuul::…infrastructure.Zuul
    depends on nodes ServiceDiscovery
    used by services coreService::com.lakeside.CustomerCore,
    used by nodes coreContainer::CustomerCoreContainer {
    default values {
        hostname = "APIGateway"
        port = 8080
        apiUri = "eureka:8080"
    }}...
```

(b)

Listing 2: Adapted operation models for resolving the security smell of Publicly Accessible Microservices with an API Gateway integration.

4 Related Work

[9] presents a series of security smells for MSA, along with suggested refactoring possibilities to resolve them. However, detecting these issues automatically and refactoring applications to mitigate them remains an unresolved challenge. As far as we are aware, the only available research on this topic is the work by [8], which proposes a trade-off analysis to determine whether implementing a refactoring is advisable based on the impact it will have on the overall application quality and the specific security concern.

There are already some methods and tools for analyzing microservices applications' security, which can also be used to detect other security smells. For instance, [12] proposes a static analysis technique to detect security smells in infrastructure-as-code [6] scripts. [12], however, differs from our proposal in its objectives, as it focuses on detecting security smells for infrastructure-as-code only, while we consider the detection and refactoring microservice security smells for different viewpoints, e.g., the service or operation viewpoint.

5 Conclusions

We have introduced a model-based approach for resolving security smells in microservices' based on extending LEMMA functionality. Our approach process extended LEMMA functionalities to automatically detect the two most recognized microservices' security smells and recommend refactoring strategies to resolve their effects.

For future work, we plan to follow the exact modeling and analysis methodology to extend the current implementation into a full-fledged prototype featuring a model-driven resolution of security smells occurring in MSA. Furthermore, we plan to include *Software Architecture Reconstruction* (SAR) [2] to automatically derive security-aware LEMMA models based on the current implementation of the software system to ease the integration of our approach in MSA development.

We also plan to exploit the full-fledged prototype to validate and evaluate our method on real-world applications, with the goal of demonstrating how our approach facilitates the development process of MSA by providing means for security smell resolution.

In this perspective, we also plan to assist developers in deciding whether/how to refactor a security smell detected in an MSA, e.g., by integrating our full-fledged prototype with trade-off analyses and code generation functionalities to resolve the security smell on the implementation level automatically. Additionally, we plan to extend our approach to work with other microservice-related smells, e.g., architectural smells.

Acknowledgements

This work was partially supported by ANID under grant ANID PIA/APOYO AFB220004.

References

[1] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, third edition, 2013.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 4th edition, 2021.

[4] Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, first edition, 2017.

[5] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption – a survey among professionals in Germany. *Enterprise Modelling and Information Systems Architectures*, 14(1):1–35, 2019. German Informatics Society.

[6] Kief Morris. *Infrastructure as code*. O'Reilly Media, 2020.

[7] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.

[8] Francisco Ponce, Jacopo Soldani, Hernán Astudillo, and Antonio Brogi. Should microservice security smells stay or be refactored? towards a trade-off analysis. In Ilias Gerostathopoulos et al., editors, *Software Architecture*, pages 131–139. Springer International Publishing, 2022.

[9] Francisco Ponce, Jacopo Soldani, Hernán Astudillo, and Antonio Brogi. Smells and refactorings for microservices security: A multivocal literature review. *Journal of Systems and Software*, 192:111393, 2022.

[10] Florian Rademacher. *A Language Ecosystem for Modeling Microservice Architecture*. PhD thesis, University of Kassel, 2022.

[11] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Deriving microservice code from underspecified domain models using DevOps-enabled modeling languages and model transformations. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications*, pages 229–236. IEEE, 2020.

[12] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175, 2019.

[13] Chris Richardson. *Microservices Patterns*. Manning Publications, 2019.

[14] Philip Wizenty, Francisco Ponce, Florian Rademacher, Jacopo Soldani, Hernán Astudillo, Antonio Brogi, and Sabine Sachweh. Towards resolving security smells in microservices, model-driven. In *2023 18th International Conference on Software Technologies (ICSOFT)*, pages 15–26. SCITEPRESS, 2023, (accepted for publication).