A software development framework for secure microservices

by

Peter Tshifa Nkomo

Thesis
submitted in fulfillment of the requirements for the degree

Doctor of Philosophy
in
Computer Science

in the
Faculty of Science

at the
University of Johannesburg

Supervisor
Professor Marijke Coetzee

January 2019

Acknowledgements

I would like to acknowledge the scholarly support and motivation that I received from my supervisor Professor Marijke Coetzee. This thesis would not have been possible without her attention to details, drive for excellence and perfection.

I would like to thank my wife Rugare for her patience, love and support while I was busy working on the thesis.

I would also like to thank my two daughters, Grace and Hannah for being patience with me during the research period.

Lastly, to God be the glory.

Abstract

The software development community has seen the proliferation of a new style of building applications based on small and specialized autonomous units of computation logic called microservices. Microservices collaborate by sending light-weight messages to automate a business task. These microservices are independently deployable with arbitrary schedules, allowing enterprises to quickly create new sets of business capabilities in response to changing business requirements. It is expected that the use of microservices will become the default style of building software applications by the year 2023, with the microservices' market projected to reach thirty-two billion United States of American dollars.

The adoption of microservices presents new security challenges due to the way the units of computation logic are designed, deployed and maintained. The decomposition of an application into small independent units increases the attack surface, and makes it a challenge to secure and control network traffic for each unit. These new security challenges cannot be addressed by traditional security strategies. Software engineers developing microservices are facing growing pressure to build secure microservices to ensure the security of business information assets and guarantee business continuity.

The research conducted in this thesis proposes a software development framework that software engineers can use to build secure microservices. The framework defines artefacts, development and maintenance activities together with methods and techniques that software engineers can use to ensure that microservices are developed from the ground up to be secure. The goal of the framework is to ensure that microservices are designed and built to be able to detect, react, respond and recover from attacks during day-to-day operations. To prove the capability of the framework, a microservices-based application is developed using the proposed software development framework as part of an experiment to determine its effectiveness. These results, together with a comparative and quality review of the framework indicate that the software development framework can be effectively used to develop secure microservices.

Table of Contents

Table of Figures

Table of Tables

PART I

Chapter 1

Introduction

1.0 Introduction

Enterprises need timely access to information to be able to retain business agility and boost productivity (Da Xu, 2014). The key to unlocking the value of existing technologies is often the ability to integrate existing applications and to assemble various technology components quickly to create new sets of business capabilities (Porter & Heppelmann, 2015). Over the years, enterprises have adopted an architectural style called service-oriented architecture (SOA), that aims to create an integrated information technology infrastructure that is scalable, and can quickly respond to changing needs (Natis & Schulte, 2003). Despite the popularity of SOA, implementing SOA in a fast-paced business environment with many new competitors frequently joining the market has proved to be a challenge. SOA-based applications are complicated to maintain and enhance in response to new business changes, and can become bottlenecks to business innovation. A new architecture called the microservices architecture, that is used to realize SOA, has emerged to enable organizations to make the development or enhancement of applications faster and easier to manage (Pahl & Jamshidi, 2016).

The microservices architectural style is based on small and specialized autonomous units of computation logic called microservices, that are independently deployed using arbitrary schedules (Newman, 2015, Lewis & Fowler, 2014). Microservices communicate using point-to-point exchanges of message by means of lightweight mechanisms over the hypertext transfer protocol (HTTP) or by listening to events within their operating environment (Dragoni et al., 2017). Research indicates that the microservices architecture is to become the default software architecture by the year 2023 (LightStep, 2018). The microservices architecture's market is projected to reach thirty-two billion United States of American dollars by the year 2023 (Infoholic Research LLP, 2017).

Although the microservices architecture constitutes an essential trend in software design with significant implications in the manner in which software is constructed, surveys such as the one conducted by Dragoni et al. (2017) highlight a general lack of comprehensive research into microservices security. On the other hand, it is estimated that the total cost of cybercrime damages will be six trillion United States of American dollars annually by the year 2021 (Morgan, 2018). In this regard, software engineers face a growing pressure to build secure applications. The challenge, however, is that a limited number of software engineers is trained in secure software development (Zhu et al., 2013). Feng et al. (2016) observes that the security of software often receives attention when security weakness or breaches are reported. Consequently, it is not uncommon that the same security issues re-occur over and over again (Veracode, 2017).

The aim of this research is to address the challenge of building secure microservices in agile software development environments with quick turnaround times. The focus is placed on the identification of security activities that can be integrated with the phases of software development frameworks and methodologies such as the Agile methodology. The aim is to develop a generic framework that can be applied to a variety of software development methodologies.

1.1 Description of the problem area

To date, no formal approach exists to secure microservices, therefore practitioners are left to guidance found in blogs, online articles, and software development conferences, where such discussions are often limited in scope and detail (Yarygina, 2018). Furthermore, research on microservices security such as by Sun, Nanda, and Jaeger (2015), Fetzer (2016), Otterstad and Yarygina (2017), Yarygina and Bagge (2018) is mostly piecemeal approaches focusing on certain parts of the microservices architecture and do not take a comprehensive view to microservices security. Although it is vital to address the security of individual components of a microservices-based application, new security challenges are likely to emerge when various parts of the application interconnect (Fernandez-Buglioni, 2013). To address the security challenge, a holistic risk analysis of the microservices architectural style is required together with a systematic approach to building microservices in a way in which security is an integral part of the entire microservices lifecycle (Fernandez-Buglioni, 2013). Microservices need to address security throughout the whole lifecycle to prevent or minimize the impact of attacks identified in a risk analysis (Feng et al., 2016). Building microservices with security in mind in this manner avoids expensive re-engineering efforts that may be required after source code is written or a security breach has occurred (De Alwisa et al., 2018).

1.2 Motivation

The microservices architecture is experiencing a market proliferation at a time when there is an increase in cybercrime. Secure software development approaches have been proposed to ensure that software is developed with fewer security weaknesses such as the Microsoft SDL (Howard & Lipner, 2006), TouchPoint (McGraw, 2006) and OWASP's Comprehensive, Lightweight Application Security Process (CLASP) (OWASP, 2006), among many others. However, these approaches are not designed for new trends in software development that many enterprises are investing in to ensure rapid software releases to retain business agility and boost productivity. On the one hand, software engineers are under immense pressure to create production-ready software applications quickly, yet on the other hand, there is no systematic guidance to assist software engineers, who often are not trained in software security, to develop secure software. Consequently, every year the top ten common security weaknesses are strikingly similar to those reported in the previous years. Software engineers can thus benefit from a light-weight software development methodology, with easy-to-use reusable security artefacts, that provide guidance on how to develop secure microservices from the start of the development process.

1.3 Problem statement

Even though the microservices architecture makes the building of complex applications easier, the management of microservices security has become more challenging. The management of the security of traditional SOA-based monolithic applications can be performed using a centralised security component that ensures that security services such as authentication, and authorisation are of high assurance. Due to the distributed nature of microservices, such a centralised security component could impact efficiency and limit the purpose of the architecture. The absence of assurance provided by centralised security, coupled with the lack of formal approaches to secure microservices, creates a need to augment software development frameworks with relevant security activities to assist software engineers when creating microservices-based applications.

1.4 Research objective

In order to address the research problem, the primary objective of this research is to propose a microservices software development methodology that can assist software engineers to quickly build secure microservices so that these microservices can detect, resist, react and recover from security attacks.

The objective of this research is met by addressing the following research questions (RQ):

RQ1 - What are the security challenges associated with the microservices architectural style?

This question is addressed by first understanding the emergence of the microservices architectural style and how it differs from traditional SOA implementations. The microservices architectural style is a relatively new area of research (Di Francesco, Malavolta & Lago, 2017). In that regard, there is a need to first understand the concepts and components of both the SOA model and the microservices architectural style. This understanding is vital to determine if protection measures used to secure traditional SOA implementations in the past are sufficient in microservices implementations. Furthermore, a risk assessment can assist to fully understand the security challenges of the microservices architectural style. Once a risk assessment is conducted, it becomes vital to analyze weaknesses of the microservices architectural style from the perspective of a potential attacker to reason better about appropriate countermeasures. To this end, the following secondary questions are defined:

a) How does the microservices architectural style differ from common SOA implementations?
b) What are the security risks of microservices?
c) What methods can an attacker use to exploit weaknesses in the microservices architecture?

RQ2 - How can software engineers build microservices in a systematic way so that security is an integral part of the entire microservices lifecycle?

The objective of this question is to ensure that software engineers do not use a reactive approach to secure microservices. The question is addressed by first understanding the design flaws that software engineers should avoid when using the microservices architectural style. This understanding is vital because a component of the system with many design flaws tend to correlate with a high number of security weaknesses (Feng et al., 2016, Mirakhorli, 2014). Once the potential design flaws are identified, appropriate design guidelines can be identified to assist software engineers to avoid subtle architecture-level security weaknesses. Also, in a fast-paced development environment, software engineers should be able to count on security-focused tools and techniques to guide the quick development of secure microservices. In this process, microservices maintenance activities should not deteriorate any current protection measures. To address this research question in more detail, the following secondary questions are defined:

a) What are the design flaws associated with the microservices architectural style?
b) What guidelines can software engineers use to design and implement secure microservices and how can these guidelines be presented in a manner that is useful and convenient for software engineers especial those not trained in software security?
c) How can security-focused tools, techniques, and practices be integrated in the development lifecycle so that they become part of the software engineer's daily software development tasks?

RQ3 - How can protection measures be correctly implemented and preserved to ensure that microservices are safe at all times?

This research question directly follows from question RQ2 stated above. While it is essential to identify security weaknesses and their countermeasures, there is also the risk that guidelines and protection measures may not be followed, or may be implemented incorrectly. An incorrect implementation may result in the introduction of new weaknesses (IEEE Center for Secure Design, 2015). Strategies should be in place to detect the avoidance of guidelines and incorrect implementation of protection measures, as not many software engineers are trained in security (Zhu et al. (2013) and are under pressure to deliver working software.

1.5 Research contributions

This thesis provides a holistic security perspective of the microservices architectural style. The study identifies the security challenges of the microservices architectural style and designs a catalogue of microservices security threats, security weaknesses, and their mitigations. Furthermore, the study designs a dictionary of coding guidelines to mitigate common microservices security weakness and common attacks on microservices. The catalogue and dictionary are provided as reusable artefacts in a manner that is easy to use for software engineers who are not trained in security. Software engineers can use the catalogue and dictionary to gain background knowledge that is required to conduct risk assessments, or as security training manuals when performing brainstorming sessions during threat modelling exercises.

Furthermore, the catalogue and guidelines can be used as a reference by software engineers in their day-to-day microservices development tasks.

In addition, a software development framework is further proposed to build secure microservices from the ground up based on the identification of security-focused activities that are required in a microservices development lifecycle. The framework is specified in a manner that makes it agnostic to both culture and technology characteristics in a software development team to allow software engineers to apply software security controls within their unique organizational circumstances.

To date, two publication has been published from this research.

Nkomo, P. T. and Coetzee, M. (2016) *Engineering Secure Adaptable Web Services Compositions*, In Proceedings of the 2016 International Conference on Information Resources Management, CONF-IRM 2016, May 18-20 Cape Town, South Africa, ISBN: 978-0-473-35594-4

Nkomo, P.T and Coetzee, M (2019) *Software Development Activities for Secure Microservices*, In Proceedings of the 19th International Conference on Computational Science and its Applications, ICCSA 2019, July 1-4 St Petersburg, Russia, Lecture Notes in Computer Science (LNCS) 11623

1.6 Research methodology

The research in this thesis commences with the formulation of a problem statement and research questions to justify the purpose of the research. The ultimate aim is to develop a systematic light-weight software development methodology that software engineers can use to develop secure microservices. The main contribution is a conceptual framework called SAFEMicroservices, that takes the form of a secure software development framework. In order to achieve this, the research is conducted using deductive reasoning and empirical research methods, and is qualitative in nature. The formulated research questions are addressed throughout the research strategy and through the development of a systematic light-weight software development methodology. The systematic light-weight software development methodology makes use of techniques and knowledge identified in the literature review (Ramesh et al., 2004).

To address the research objectives, the research questions that are posed are investigated further over the course of this thesis to gain an understanding of the problem domain and to provide a sound motivation for the solution proposed for the research problem. In order to achieve this, a scientific and well-defined research methodology must be followed to ensure the soundness of the results found during the course of this work. The research makes use of a strategy discussed in Olivier (2009). The strategy discussed in Olivier (2009) consists of a detailed investigation of the literature, followed by an in-depth analysis and review. The findings of the review are used to motivate the proposed framework. Finally, the framework is evaluated. Each of these methods are now described in more detail.

Literature review:

This research conducts a literature review of service-oriented architecture, microservices, web services security, and security threats and risks, tools and techniques that can be used in the development of secure software. This ensures that a solid understanding is gained of all concepts that contribute to every facet of the problem and provide the basis for a complete solution. The current state-of-the-art is determined through the review of the literature, and also, the limitations of existing methods, techniques, and approaches used in the past to address the problem statement are ascertained. The literature review is conducted in Chapters 2 to 6 to determine the current state of the microservices security to gain insight into existing methods, techniques, and approaches which have been previously proposed to address the problem statement. From the literature review a formal foundation for the proposed solution is defined by the sets of security requirements, microservices specific threats and security activities that are derived.

Framework:

The formal foundation for the proposed framework is found by the literature review. A basis is provided to create a software development framework that guides software engineers to identify relevant risks and threats and to apply protection measures comprehensively. The security framework aims to define a systematic way of doing things in the microservices architecture environment, following secure software engineering principles to incorporate system security features during development. The SAFEMicroservices framework presents an innovative technique that supports current standards, policies and procedures, best practices and supporting tools. The framework is proposed by the researcher in Chapter 7 to address the problem statement.

Evaluation:
The evaluation of the framework is performed according to a formal and rigorous approach. Due to the nature of the software development framework, it would be very difficult to determine its real value as perceived by industry. A real-world evaluation would be complex and costly to implement. Such a framework can only prove its real value after being used for many years in industry. In order to comprehensively evaluate the conceptual framework, two types of evaluation are performed. Firstly, an implementation of an example application is presented and reviewed. This proof-of-concept implementation is used to determine if the proposed software development methodology can adequately be used to develop secure microservices. The evaluation criteria are identified in the review and analysis of literature. Secondly, a comparative evaluation of the framework is done with regards to the functionality that it provides, to determine if it does make a contribution. Finally, the framework quality is reviewed by using a well-established quality model, where quality is determined by a number of evaluation criteria.

Finally, the researcher presents a critical evaluation to determine if the framework meets the requirements initially identified for this research. Additional arguments are made as to the relevance of the solution and if software engineers will adopt the proposed solution.

1.7 Thesis outline

This thesis consists of four parts namely Part I, Part II, Part III, and Part IV. Each part consists of several chapters. Figure 1.1 below depicts the layout of the thesis. Part I provides the required background and review of the literature and is essential to the formulation of the proposed systematic light-weight software development methodology. Essential concepts are presented in this part of the thesis and provide the building blocks of the proposed software development methodology. Part I consist of Chapter 1, Chapter 2, Chapter 3 and Chapter 4. The current chapter, Chapter 1, presents the research topic and introduces the problem that the thesis attempts to address.

Chapter 2 defines SOA and review the SOA model. This chapter aims to understand SOA as the foundation on which the microservices architecture is constructed. The technologies that enable SOA are discussed, and the chapter also describes how units of computation logic called services are combined to automate enterprise business tasks. Lastly, the challenges of SOA are identified.

Chapter 3 presents the state-of-the-art in microservices and the microservices architectural style. The principles and concepts behind the architecture are explained. Lastly, the general security challenges of the microservices architecture are presented.

Chapter 4 provides a background to understanding the security of both traditional SOA implementation and the microservices architecture implementation. General security concepts are defined in this chapter. Also, the chapter discuss how security is implemented in SOA and microservices.

Part II of the thesis discusses the security risk assessment of microservices. An analysis of the risks associated with microservices culminates in the identification of activities that should be part of the development lifecycle of microservices. Part II of this thesis consists of Chapter 5 and Chapter 6.

Chapter 5 discusses a preliminary security risk assessment of microservices. The chapter aims to identify weaknesses in microservices and harm that may arise from misuse of microservices by a malicious user. The chapter concludes by identifying security-focused activities that can assist in developing secure microservices.

Chapter 6 identifies and reviews available security-focused tools and techniques that can be used to perform security-focused activities identified in Chapter 5. This chapter aims to understand how software engineers can adopt tools and techniques as part of their daily development activities.

Part III of the thesis proposes a software development framework that can be used to develop secure microservices. The methodology is demonstrated by developing an example application. The framework is then evaluated. Part III consists of Chapter 7 and Chapter 8.

Chapter 7 discusses the proposed systematic light-weight software development framework that can be used to develop secure microservices. The various reusable artefacts of the methodology are introduced.

Chapter 8 demonstrates the applicability of the framework proposed in Chapter 7 in practice. An example application is used to demonstrate the feasibility of using the proposed framework.

Part IV of the thesis evaluates the proposed methodology and provide concluding remarks. Part IV consist of Chapter 9 and Chapter 10.

Chapter 9 evaluates the proposed light-weight methodology using evaluation criteria identified in the review of literature.

Chapter 10 presents the conclusions of the thesis. Research contribution and future research directions are discussed.

Appendix presents a list of all the artefacts produced in this thesis.

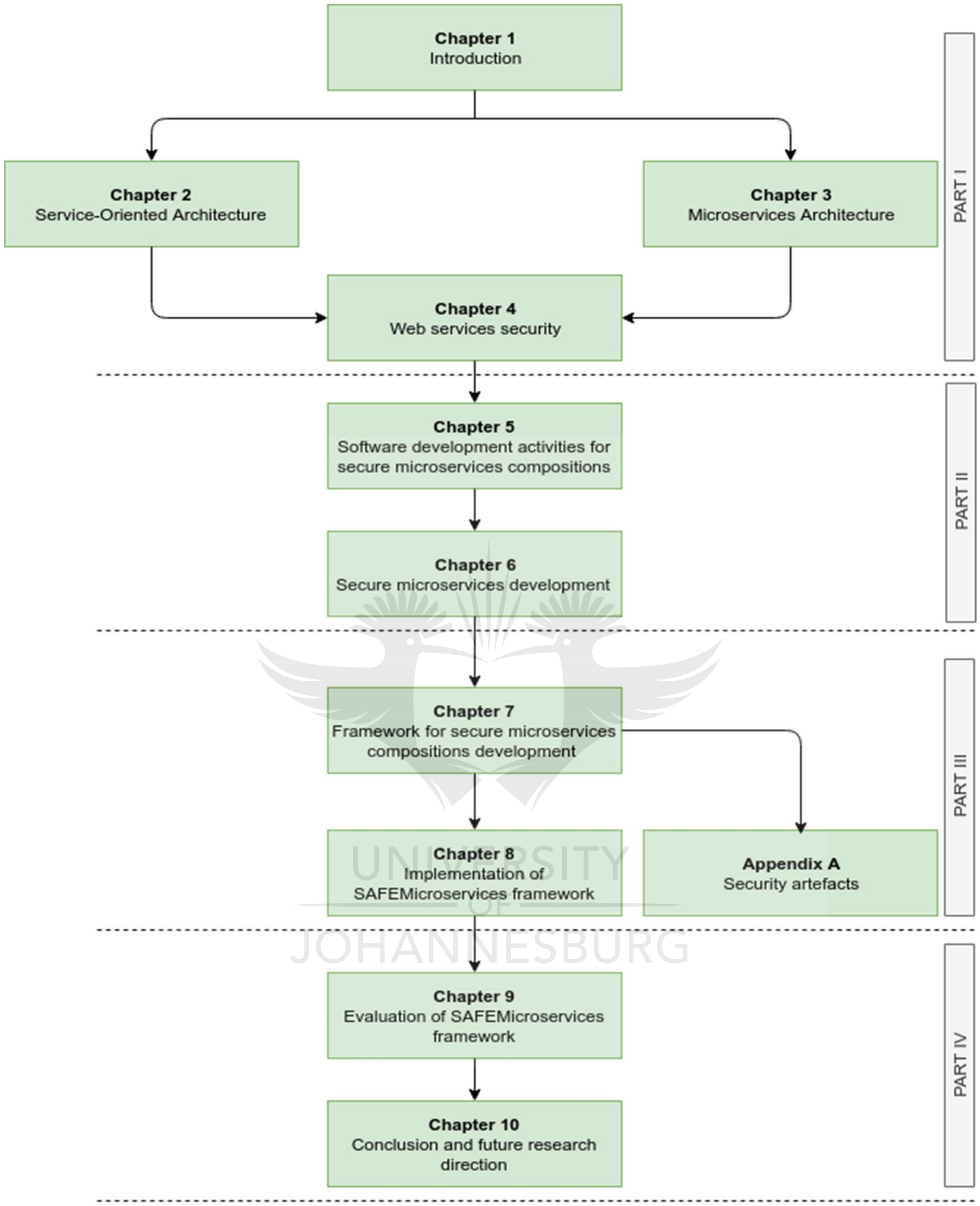

Chapter 1
Introduction
Chapter 2
Service-Oriented Architecture
Chapter 3
Microservices Architecture
Chapter 4
Web services security
Chapter 5
Software development activities for secure microservices compositions
Chapter 6
Secure microservices development
Chapter 7
Framework for secure microservices compositions development
Chapter 8
Implementation of SAFEMicroservices framework
Appendix A
Security artefacts
Chapter 9
Evaluation of SAFEMicroservices framework
Chapter 10
Conclusion and future research direction
PART I
PART II
PART III
PART IV

Figure 1.1. Thesis layout

Chapter 2

Service-Oriented Architecture

2.0 Introduction

Enterprise information systems can be composed of a variety of legacy, custom and third-party applications (Markus & Tanis, 2000). As an enterprise grows, the need for timely access to information becomes imperative to be able to retain business agility and boost productivity (Da Xu, 2014). Enterprises need to harness different types of technologies to fend off business disruption, compete against new market entrants, create new revenue streams, and meet customer demands. The key to unlocking the value of existing technologies is often the ability to integrate existing applications and to assemble various technology components quickly to create new sets of business capabilities (Porter & Heppelmann, 2015). To this end, it becomes vital for an enterprise to adopt an architecture that leads to highly flexible and maintainable systems that can continuously adapt to new business requirements (Krafzig, Banke & Slama, 2005). Over the past number of years, the need to integrate enterprise applications has led to the emergence of many integration patterns. The natural progression of the various integration efforts has culminated into an architectural style called Service-Oriented Architecture (SOA), that aims to create an integrated information technology infrastructure that is scalable, and can quickly respond to changing needs (Natis & Schulte, 2003).

This chapter aims to review the SOA model, understand its enabling technologies and to identify its challenges. The understanding gained in the review is vital towards answering research question RQ1. First, various integration patterns adopted before SOA are briefly reviewed as background. Then, Service-Oriented Architecture (SOA) is discussed by giving an overview of various enterprise integration patterns adopted in the past before the adoption of SOA in Section 2.1. Section 2.2 introduces and discuss SOA concepts and enabling technologies. Section 2.3 discusses how essential components called services, the main building blocks of SOA, are classified. Section 2.4 describes how services are combined to automate enterprise business tasks. Section 2.5 discusses the challenges of implementing SOA. A summary and conclusion then follow.

2.1 Enterprise integration

In the past, the need to integrate enterprise applications has led to the emergence of many integration patterns. The earliest integration pattern was the sharing of data, where applications produced files to be directly consumed by other applications (Ray, 2010). File sharing challenges identified were: a standard file format was required for all applications; the output of one application rarely produced what another application needed thus requiring additional processing; and file transfers lacked timeliness (Hohpe & Woolf, 2004). To compensate for file sharing challenges, the sharing of information via enterprise databases later became a more familiar pattern (Hohpe & Woolf, 2004, Vernadat, 2007). However, effective sharing of enterprise databases required standardization and governance across systems, which was challenging when systems were owned by different teams or organizations (Hohpe & Woolf, 2004). To address the need for real-time data sharing, remote procedure calls (RPC) was adopted that enabled one computer to call a procedure on another computer while abstracting the socket layer (Olsson & Keen, 2004). Unfortunately, remote procedure calls did not offer code reuse as the logic for network communication was embedded in the client and server applications. Furthermore, the client in an RPC model of communication was required to wait for a server to respond before proceeding (Ni & Yuan, 1996).

Asynchronous messaging, typically enabled by Message-Oriented Middleware (MOM) technology evolved to address the limitations of RPC for integration (Menasce, 2005). In asynchronous messaging, applications communicate by sending and receiving messages from a buffer called a queue, and no dedicated communication link is established between applications. Later, the emergence and ubiquity of the Internet and the World-Wide Web provided web-based technologies that could be used as basic building blocks for enterprise integration (Linthicum, 2003, Newcomer & Lomow, 2005). By leveraging MOM functionality and web technologies and protocols, a design paradigm that can combine reusable, coarse-grained components was adopted called *Service-Oriented Architecture* (SOA) (Natis & Schulte, 2003).

The next section defines Service-Oriented Architecture by giving a definition and describing the concepts service-orientation and Service-Oriented Architecture.

2.2 Service-Oriented Architecture

Service-Oriented Architecture (SOA) can better be understood by first describing *service-orientation* as underlying concept. *Service-orientation* is a design approach that aims to create individually constructed units of logic in a way that allow the units of logic to be collectively and repeatedly utilized to realize a specific business goal (Erl, 2008). These units of logic are called *services* (Erl, 2008). A service-oriented environment is based on a vendor-neutral architectural model, allowing an enterprise technology landscape to evolve in line with business requirements without being limited to the characteristics of a proprietary platform (Erl et al., 2014). To achieve service-orientation, the units of logic should have the following characteristics:

- Be assembled and reconfigured effectively in response to changing business requirements (Dhara, Dharmala & Sharma, 2015).
- Support a standardized contract for communications that hides underlying technology disparities, allowing each unit of logic to be individually governed and evolved (Erl, Merson & Stoffers, 2017).
- Exist within a business-centered functional context, to allow units of logic to mirror and evolve with business requirements. Each unit of logic is delivered and viewed as an asset that is expected to be reused in different business contexts (Erl, Merson & Stoffers, 2017).

The concept of service-orientation is the foundation of the service-oriented architecture. SOA represents an architectural style that aims to create an integrated information technology infrastructure that is scalable, reliable, and responsive to the changing business requirements of an enterprise (Shirley et al., 2012). SOA positions units of logic called *services* as the primary means through which an integrated information technology infrastructure can be realized (Erl, 2005) and defines how various systems within the entire enterprise interact (Wolff, 2016). Implementations of SOA are traditionally a combination of technologies, products, application programming interfaces and supporting infrastructure (MacLennan & Van Belle, 2014) ensuring that a deployed SOA tends to vary from one enterprise to another. As an architectural model, SOA encompasses the following (Erl, 2005):

- *Service architecture* - the architecture of a single unit of logic or service.
- *Service composition architecture* - the architecture of many aggregated units of logic working together to perform a business function.

The next two sections discuss *service architecture* and *service composition architecture* respectively.

2.3 Service architecture

A *service* is a unit of logic deployed as a physically independent software program with specific design characteristics that supports a business goal (Erl, Merson & Stoffers, 2017). Each service is designed within a specific business-related functional context and is comprised of a set of capabilities related to the functional context. A service is, therefore, a container of a set of related functions called *service capabilities* (Erl, 2008) that are accessed using standardized interfaces. Figure 2.1 below depicts a *Driver Service* from an on-demand taxi application such as the one provided by Uber (Cramer & Krueger, 2016). The service exposes capabilities to retrieve a list of available drivers, add new driver and remove a driver from the system.

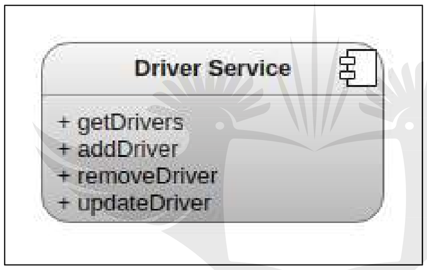


Figure 2.1. Driver service with multiple capabilities

The most widely used services in SOA are based on eXtensible Markup Language (XML) and JavaScript Object Notation (JSON). XML is a self-descriptive markup language designed to store and transport information (Moller & Schwartzbach, 2006). JSON is a lightweight, text-based, language-independent data interchange format (Crockford 2006). Services that are based on XML and JSON are referred to as *web services* (Dhara, Dharmala & Sharma, 2015, Pautasso, 2014). Web services have become an essential means to implement SOA (Ochieng et al., 2011). To this end, the next section discusses web services in more details.

2.3.1. Web services

A w*eb service* is a software system identified by a Uniform Resource Identifier (URI), that has public interfaces and bindings defined and described using XML or JSON (Dhara, Dharmala & Sharma, 2015). A URI is a string character that identifies a resource. Interaction with a web service is performed in a manner prescribed by the web service definition. The web service definition is given using XML or JSON messages, and messages are conveyed over the HyperText Transfer Protocol (HTTP) (Fielding, 1999). Public interfaces are standardized communication interfaces which act as service contracts that form an interaction agreement between a web service and its consumers. The service contract is a fundamental part of the service architecture as the contract definition gives the web service a public identity and expresses the web service's functional context (Erl, 2008).

In the context of SOA, it is important to note that merely using web services does not necessarily translate to a SOA implementation, and not all SOA implementations are based on web services. However, the relationship between SOA and web services is essential and is mutually influential (Ochieng et al., 2011, Candido et al., 2013). When web services are used to implement SOA, the implementation represents a web services-based implementation of SOA. In a web-services-based SOA implementation, the web services architecture becomes vital.

There are two popular architectural styles currently in use in web services architecture, namely the Simple Object Access Protocol (SOAP) and Representational State Transfer (REST). The two web service architectural styles are discussed next.

a) SOAP

SOAP is a lightweight protocol that exchanges structured information in a distributed environment using XML technology (Box et al., 2004). The SOAP specification defines the standard message format used by most web services implementations. SOAP originally stood for "Simple Object Access Protocol" but is now considered a standalone term. The SOAP message is contained in an *envelope* (Box et al., 2004). Inside the envelope is an optional *header* element and *body* element. The header is used to hold extra meta-information about the message or any security information. The body of the SOAP message contains the payload that carries incoming or outgoing information. Figure 2.2 below shows the basic structure of a SOAP message which includes the envelope, header, and body.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
       <env:Header>
    ...
       </env:Header>
       <env:Body>
    ...
       </env:Body>
</env:Envelope>
```

Figure 2.2. Basic SOAP Message

In a SOAP-based web services architecture, a Web Services Description Language (WSDL) document provides an XML grammar to describe web services (Chinnici et al., 2007). A WSDL document describes a web service as a set of uniform resource indicators (URI) that consume XML request messages and provide XML response messages. The WSDL document describes the physical address, the network protocol and message format used by the web service. Figure 2.3 below shows the structure of a WSDL document. The *interfaces* element represents the web service interface that may contain multiple *operations*. The *service* element represents one or more uniform resource indicators (URI) through which the web service can be accessed. *Messages* represent collections of input or output parameters and can contain multiple parts that represent either incoming or outgoing information. The *binding* element associates protocol and message format information to the functions of a web service.

```
<definitions>
    <interface name="">
    ...
    </interface>
    <message name="">
     ...
    </message>
    <service>
    ...
    </service>
    <binding name=" ">
    ...
    </binding>
</definitions>
```

Figure 2.3. Structure of a WSDL document

The web service architecture defines a central directory to host the WSDLs so that the web services can be discovered. The Universal Description, Discovery Integration (UDDI) (Liu et al., 2005) provide such mechanisms (World Wide Web Consortium, 2003). UDDI allows web services to advertise themselves.

UDDI is the broker component of web services architecture that allows service providers and requestors to locate each other. UDDI is usually implemented as a distributed database with interconnected servers. Figure 2.4 below adapted from Erl (2004) depicts the relationship between SOAP, WSDL, and UDDI.

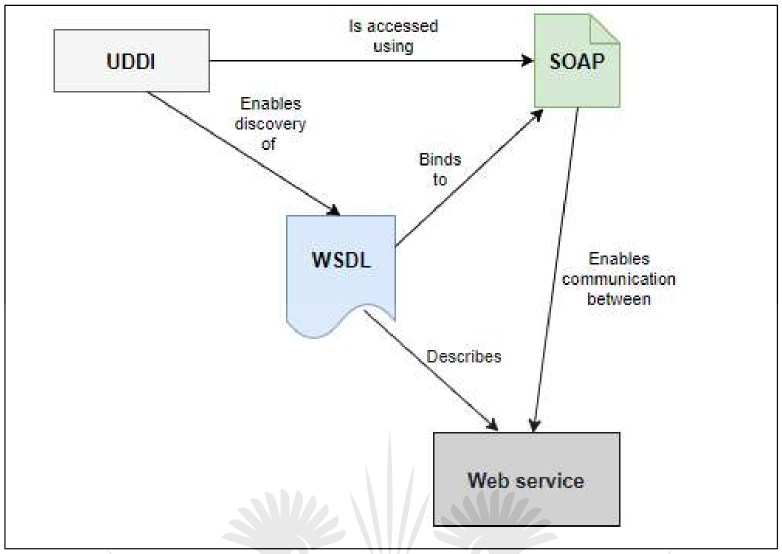


Figure 2.4. The relationship between SOAP, WSDL, and UDDI

In Figure 2.4, the provider of a web service describes the web service using a WSDL document and optionally publishes it to a UDDI repository. A web service consumer queries the repository to locate a web service and is sent the WSDL document of the service. The WSDL document can also be retrieved from the service provider out of band using, e.g. an email message. The WSDL document describes the format of requests and responses that the web service provider expects and provides respectively. All messages sent between the service provider, service consumer and repository are sent using SOAP.

Next, the REST architectural style used in the web services architecture is discussed.

b) Representation State Transfer (REST)

REST is a software architectural style that supports the transmission of data using the Hypertext Transfer Protocol (HTTP) (Fielding, 2000). A REST web service is represented as a resource and exposed with a uniform resource identifier (URI). REST defines how the state of resources are addressed and transferred over HTTP by a wide range of clients written in different languages. Figure 2.5 shows an example of a request to access a resource using the RESTful style.

```
GET --header ‘Accept:application/json’
             ‘https://localhost:8084/service/user/5678/details’
```

Figure 2.5. REST request

REST defines four basic architectural design principles (Fielding, 2000) as follows:

- HTTP methods are used explicitly. For example, in Figure 2.5, the resource is retrieved using the HTTP *GET* method. The GET method is used to request the representation of a specified resource following the protocol defined in the request for comment RFC 2616 (Fielding et al., 2009).
- A uniform resource identifier (URI) with a structure similar to directories is exposed. The URI should be a self-documenting interface which can be intuitively understood without requiring any explanation or reference. In Figure 2.5 the URI is */service/user/5678/details’,* which defines the document type resource.
- Data transfer should use XML, JavaScript Object Notation (JSON), or both. JSON is a lightweight data-interchange format that is self-describing and easy to understand. JSON is built using two structures namely a collection of name-value pairs and an ordered list of values. JSON is text only and can quickly be sent to and from a server to be understood by any programming language. Figure 2.6 shows an example of a JSON response received when a request in Figure 2.5 above is processed.
- A REST web service should be stateless (Christensen, 2009). A service consumer includes all the data that is needed for the request to be fulfilled in the HTTP headers and body of a request, so that no information is stored on the server to generate a response.

```
{
      "name":"John",
      "surname":"Smith",
      "userid":"5678",
      "roles":[
      {
            "id":1
            "name":"agent"
            }
      ]
}
```

Figure 2.6. Example of JSON response

REST has emerged in the last few years as the predominant web service design model due to its adoption and use. Its simple style has made it a better choice for software engineers than SOAP web services (De Giorgio, 2010).

In any SOA-based initiative it is vital to understand different types of services, and how the web service types are effectively communicated to stakeholders in an organization. The next section discusses the primary classification of services in SOA.

2.3.2 Service taxonomy in SOA

Services in SOA are classified according to the role the service plays in the overall architecture. This formal classification is called a *service taxonomy* (Richards, 2015). There are four basic types of services namely *business services, enterprise services, application services, and infrastructure services* (Bean 2009, Marks & Bell, 2008), as shown in Figure 2.7. The types of services are defined below.

- *Business services* are abstract high-level coarse-grained services that define the core business operation. Business services provide no implementation or protocol details (Richards, 2015). They are represented using WSDL or Business Process Execution Language (BPEL). Business services are required for the successful completion of business processes.
- *Enterprise services* implement the functionality of business services and are concrete coarse-grained services. A middleware component is used as a bridge that provides an abstraction between the business services and corresponding enterprise services. Example of an enterprise

service can be a service to retrieve customer details. The service encapsulates functionality that can be shared by many applications within the enterprise. Consequently, the use of enterprise services in SOA is based on the concept of sharing (Richards, 2015). Although sharing reduces duplication, there is often a considerable penalty in the form of tight coupling. This increases the overall risk of making changes to enterprise services since these services are globally available to the organization. Changes to enterprise services require regression to test all possible uses of enterprise services within the organization, to ensure that the change does not affect existing functionality.

- *Application services* are usually specific to the context of a given application and are therefore fine-grained. The functionality provided by this type of a service is not found at an enterprise level but specific to an application within an enterprise. For example, a service to calculate de-merit points for drivers enlisted in the on-demand taxi application is an example of an application service since it will be specific to an application and not the whole enterprise.
- *Infrastructure services* are shared services that do not represent business functionality but are used to provide additional functionality such as logging, auditing, monitoring, and security.

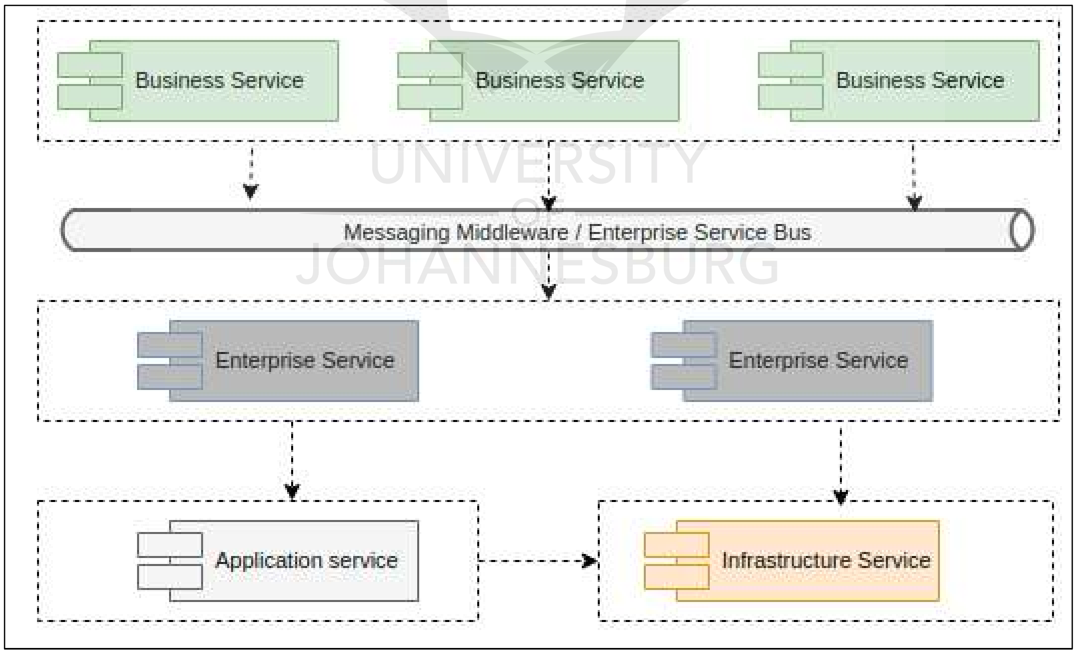


Figure 2.7. SOA services taxonomy

The classification of services results in a service ownership model that spans multiple administrative boundaries (Brown et al., 2014). Service ownership is vital to understand because it affects how teams should be coordinated to implement a SOA-based application successfully. The next section discusses the ownership of services in SOA.

2.3.3. Service ownership model

Services in SOA are owned and maintained by different service providers, spanning different administrative boundaries (Brown et al., 2014). An owner of a service can be defined as a group within the organization that has the responsibility of developing and maintaining the service. *Business services* are usual owned by business users, and *enterprise services* are owned by shared services teams such as systems architects. *Application services* are usually owned by the application development teams, and the *infrastructures services* are owned either by application development teams or a team responsible for infrastructure (Richards, 2015). Integration architects typically own the middleware components. Figure 2.8 shows the ownership model of services in SOA.

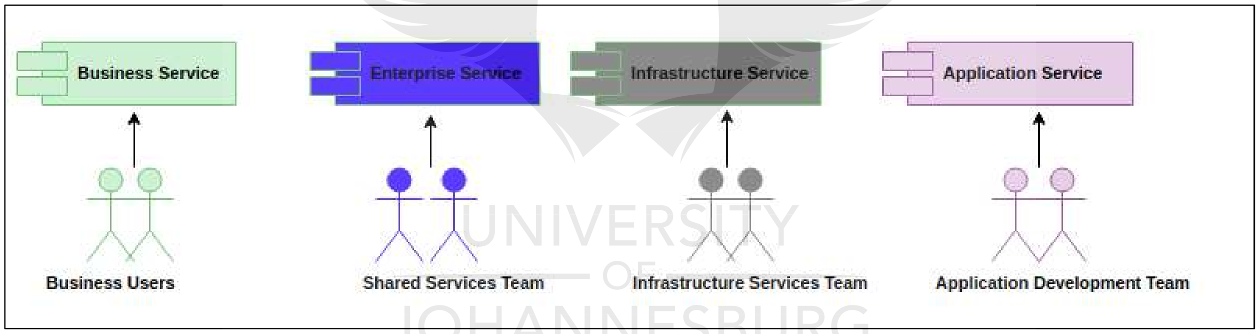


Figure 2.8. SOA Services Ownership model

In any SOA implementation, the service ownership model requires coordination among multiple groups to create or maintain applications. Any enhancement of the SOA-based application requires business users to be consulted about the abstract business services, shared services teams to be consulted about the enterprise services created to implement the business services. Furthermore, application development teams should be coordinated so that enterprise services can invoke lower-level functionality, and infrastructure teams should be coordinated to ensure nonfunctional requirements are met through the infrastructure services. Furthermore, input is required from the middleware teams or integration architects managing the messaging middleware. The effort required to develop, test, deploy, and maintain services should, therefore, be considered when migrating to SOA-based implementations.

An important design principle in service-orientation is to ensure that services are designed so that they can be effectively assembled and reconfigured to meet business requirements. The principle is important because the automation of many business tasks require a collaboration of services. When services work together to perform a business function, a *service composition* is formed (Hoffmann & Weber, 2014), discussed next.

2.4 Service composition architecture

Service-orientation requires that services are designed to be flexible logical units that can participate in aggregated structures. Aggregating of services in SOA enable complex business tasks to be automated. Even though service compositions are comprised of services, it is the service capabilities that are individually invoked and executed to carry out the function of the *service composition* (Erl, 2008). To qualify as a *service composition*, at least two participating services and a service composition initiator need to be present, otherwise the service interaction becomes a point-to-point exchange (Sheng et al., 2014). In SOA, service compositions are created through service orchestration, discussed next.

2.4.1 Orchestration of services

The service responsible for composing other services assumes the role of a *composition controller*, and composed services become *composition members* (Erl, 2008). Creating composition with a *composition controller* and *composition members* is called *orchestration* (Sheng et al. 2014). The composition controller coordinates asynchronous interactions between composition members and support sophisticated and complex service composition logic that can result in long-running runtime activities. When there is no composition controller, services interact using point-to-point exchanges within a *choreography* (Sheng et al., 2014). Figure 2.9 shows an example of an orchestration of services to automate a business task to get loan quotations. The loan request service assumes the role of composition controller and coordinates interactions with the credit scoring service, calculate interest rate service, and amount eligible service. Service orchestration is fundamental to the successful implementation of SOA.

Services compositions need to manage many types of scenarios that may arise at runtime. Therefore, the design of services as candidates for orchestration requires that services are well prepared to participate in complex service compositions (Alferez & Pelechano, 2013). Features such as security, transaction management, reliable messaging and message routing should form part of the specification of a typical service composition architecture (Erl, 2008). In general, many service compositions are dedicated to the execution and maintenance of complex business processes (Josuttis, 2007). A business process can be defined as a set of activities that once completed, accomplish a goal such as deliver a service or product to a consumer (Bhattacharya et al., 2009).

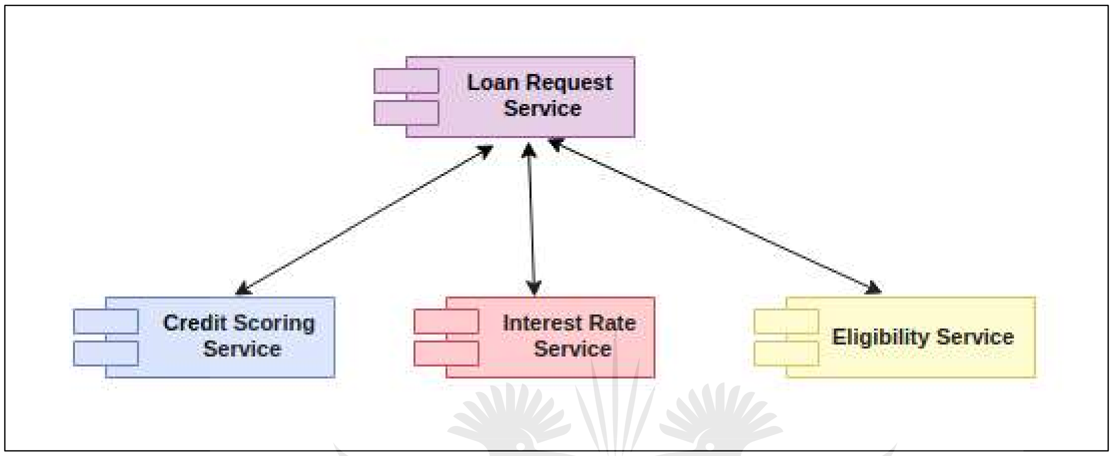


Figure 2.9. Orchestration of services

The next section briefly discusses how services are orchestrated to perform different activities that make up a business process.

2.4.2 Service orchestration and business processes

A graphical representation of a business process is created using Business Process Management Notation (BPMN) (White & Bock, 2011). BPMN is a standard for business process modeling and provides a graphical notation for specifying business processes using flowchart technique. The graphical business processes created using BPMN are then transformed to be executed using various business process modeling tools such as Business Process Execution Language (BPEL) (Pant & Juric, 2008). Conceptually, BPEL is an XML language for describing business flows and sequences. A business process, as seen by BPEL, is a collection of coordinated service invocations and related activities that produce a result, either within a single organization or across several (Parsley, 2005). BPEL is, therefore, an example of an orchestration language used to create service composition. BPEL provides many constructs to support common tasks. Examples include:

- *<invoke>* used to invoke web services

- <*receive*> used to wait for the request sent to a web service
- <*assign*> used to manipulate data variables in a process
- <*sequence*> used to define a set of activities in a process invoked in an ordered sequence
- <*flow*> used to define a set of activities that will be invoked in parallel
- <*while*> for defining loops, etc.
- <*partnerLinks*> for defining web services that are invoked by the process

The next section discusses an example to illustrate the various SOA concepts discussed above. The example uses BPEL to create a service composition utilizing the BPEL constructs discussed above.

2.5. Motivating example

Consider an imaginary on-demand taxi application such as Uber (Rogers, 2015) that is implemented as an SOA-based application as shown in Figure 2.10. The application is referred to as the *PickMeUp* application.

Registered passengers request taxi rides using mobiles phone or desktop computers. As soon as the request is made, a notification about location and passenger details is sent to the nearest driver. The driver either accepts or rejects a request for a ride. In case the ride is rejected, a notification is sent to drivers in the area. If the driver accepts the ride, driver details are sent to the customer along with the estimated arrival time. The passenger can track the drivers and drivers can track the exact location of the passenger to reach their exact location. The payment procedure between the passenger and the driver is either cash or credit card.

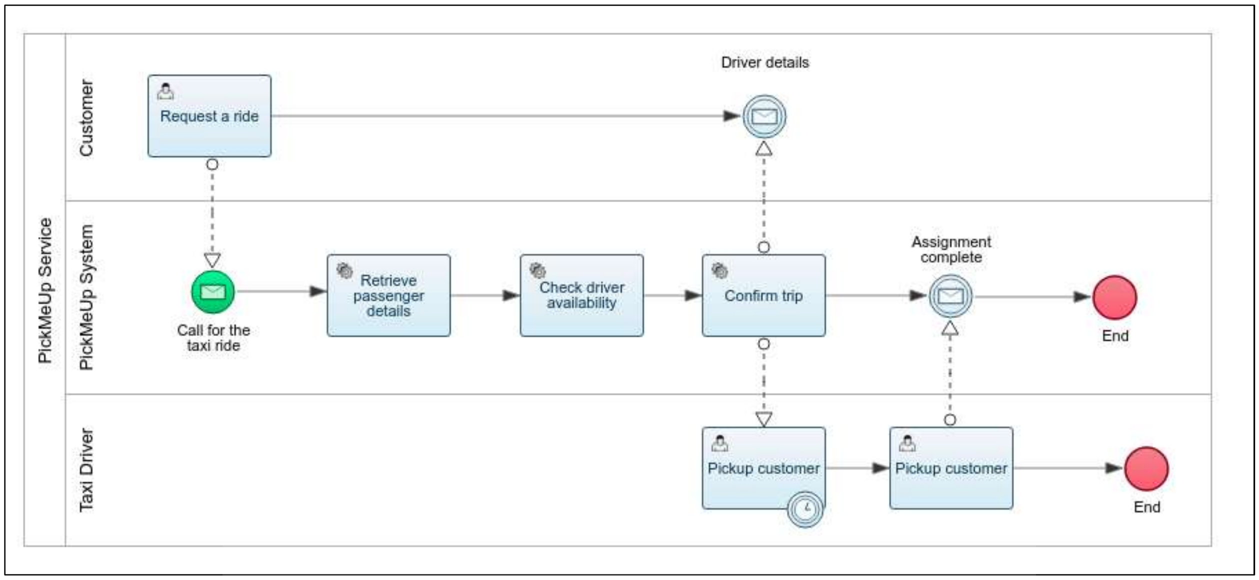


Figure 2.10. BPMN for the PickMeUp application process

In a typical SOA implementation, the business analyst uses the Business Process Management Notation (BPMN) to visualize the *PickMeUp* business process graphically. The BPMN process diagram is used by the software developer to create an executable process using BPEL. BPEL is used to orchestrate web services into an SOA services composition. The composition initiator is a service called *TaxiService*. The *TaxiService* orchestrates various business services such as the *LocationService* for retrieving the location of the passenger requesting the service, the *DriverService* to get the information about the driver who is dispatched to the passenger, the *PaymentsService* to bill the client and the *VehicleService* to manage registered vehicles. The *TaxiService* is executed in a BPEL engine such as ActiveBPEL. ActiveBPEL is a commercial-grade open source implementation engine for BPEL (Qian et al., 2007). To external clients of the *PickMeUp* application, the *TaxiService* is exactly like any other web service and is described using a WSDL and consumed using SOAP messages.

Figure 2.11 show an example of a BPEL service composition for *TaxiService*. The *TaxiService* orchestrates the *VehicleService*, *LocationService* and *DriverService*. The orchestrated services are shown as part of the <*partnerLinks*> elements.

```
<?xml version="1.0"?>
<process name="TaxiService" ..>
   <variables>
 --
   </variables>
   <flow>
--
   </flow>
   <partnerLinks>
      <partnerLink name="registeredVehicleService"
        partnerLinkType="vhs:vehicleLT"
        myRole="registeredVehicle"
        partnerRole="VehicleService"/>
      <partnerLink name="mapService"
        partnerLinkType="loc:locationLT"
        partnerRole="LocationService"/>
      <partnerLink name="registeredDrivers"
        partnerLinkType="driv:driverLT"
        myRole="registeredDriver"
        partnerRole="DriverService"/>
---
      </partnerLinks>
</process>
```

Figure 2.11. BPEL process for PickMeUp application

The orchestrated business services employ an Enterprise Service Bus (ESB) as an integration hub that mediates the invocation of web service to complete a business transaction. The ESB provides additional capabilities such as augmenting request information when necessary, modifying the format of data from one type to another, and transforming a request from one protocol to another. Various enterprise services are defined to provide functionality for business services. Security and logging services are provided as infrastructure services. Figure 2.12 shows the SOA application depicting the various concepts discussed above.

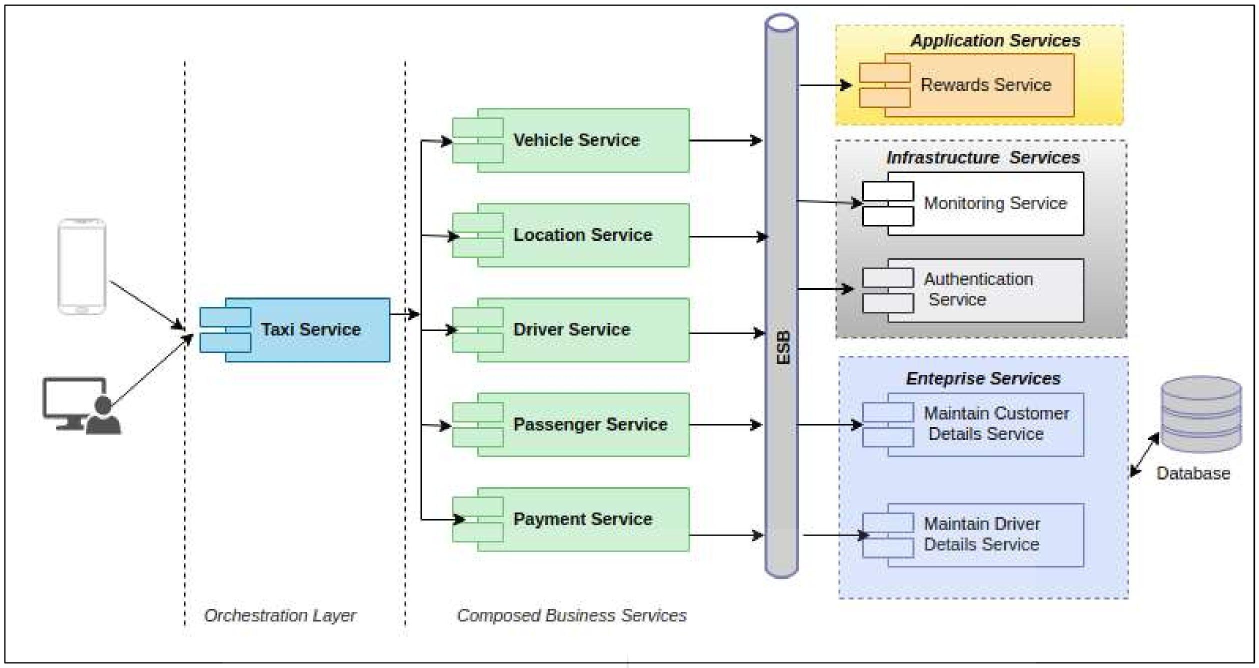


Figure 2.12. PickMeUp SOA application

In the next section, the challenges of developing an SOA-based application is described.

2.6 Implementation challenges of SOA

Implementing SOA in an enterprise has many challenges. To start with, there has been a lack of consensus on how to implement SOA correctly (Newman, 2015). The existing narrative on SOA implementations has often been provided by technology vendors whose aim is to sell their technology products (Lee, Shim & Kim, 2010). Furthermore, there is a lack of guidance on how to define the granularity of services in SOA, and how to ensure that services are not tightly coupled. Software engineers are expected to design and implement services and service compositions that meet the qualities that SOA stakeholders expect, although there is no blueprint to guide them on how to partition a complex application into a set of collaborating services. Moreover, there is often a misconception that legacy systems can easily be integrated into SOA without taking into consideration technical constraints of the legacy components, such as immature technology, that may require significant rework (Lewis et al., 2007). This often poses a risk to the adoption of SOA.

Successful deployments of SOA-based applications require extensive coordination among various stakeholders as identified in the SOA service ownership model (Bell, 2008). Multiple reviews and approvals are vital among the various groups who own business services, enterprise services and infrastructure services (Richards, 2015). Furthermore, changes to some enterprise services may have a substantial ripple effect with regards to regression testing (Bhuyan, Prakash & Mohapatra, 2012). Typically, many SOA-based applications require a history of an extensive suite of regression tests (Bartolini et al., 2011). This increases the time required for testing and the personnel required to complete testing.

SOA services compositions can become complicated. When this complexity arises, it is not always intuitive to know in an orchestration which part of the application to modify when new business requirements arise. As a result, software engineers tend to be hesitant to make enhancements because of the fear of causing damage to existing SOA-based applications due to unknown dependencies (Richards, 2015). Consequently, the rate of enhancement and delivery of new business capabilities is reduced. Furthermore, when a service composition becomes complex, new software engineers on the team require much time to become familiar with the software source code and other infrastructure components. This may increase the software project delivery timeline.

The implementation challenges of SOA have been the major pitfall towards the adoption of the architecture. Many enterprises compete in a fast-paced business environment with a lot of new competitors frequently joining the market, and can, therefore, not afford slow software releases.

2.7 Conclusion

Enterprises are often expected to harness different types of technologies to create new revenue streams and meet customer demands. The solution to harnessing technologies is usually to integrate existing applications and to assemble various technology components quickly to create new sets of business capabilities. SOA became an architecture that promised highly flexible and maintainable systems that can continuously adapt to new business requirements. The basic building blocks of SOA is web-based technologies and protocols. Using services as its foundation, SOA enables an integrated information technology infrastructure that is scalable, reliable, and can quickly respond to changing needs of an organization.

Despite the popularity of SOA, implementing SOA in a fast-paced business environment with many new competitors frequently joining the market has proved to be a challenge. SOA-based applications are complicated to maintain and enhance in response to new business changes and can become bottlenecks to business innovation. The stiff competition enterprises are exposed to requires investment in fast-paced software development environments with quick software release cycles to stay ahead of competitors. The elaborate service ownership model of SOA does not readily support fast-paced software development teams due to the requirement for extensive coordination to make software changes or enhancements.

The need to harness enterprise technologies to meet customer demands quickly require an approach that makes development or enhancement of applications faster and easier to manage. The challenges of incorporating new requirements in complex SOA applications have led enterprises to consider adopting an architecture which allows for fast and flexible development and provisioning of business processes. A new architecture called the microservices architecture can realize SOA and has emerged to enable organizations to make development or enhancement of applications faster and easier to manage. To differentiate microservices architecture from the SOA discussed in this chapter, the implementation of SOA discussed in this chapter will be referred to as the *traditional SOA implementation*. In the next chapter, microservices architecture is discussed.

Chapter 3

Microservices Architecture

3.0 Introduction

Nowadays, enterprises operate in a fast-paced business environment where competition is fierce. To be able to maintain their competitive advantage, enterprises invest in fast-paced software development technologies that support speedy software releases (Lesser & Ban, 2016). The state of software development and software operations report of 2017 found that companies that excel have forty-six times more frequent software deployments than others (Forsgren et al., 2017). These companies have four hundred and forty times faster lead time from the moment software source code is committed to a software repository to when the source code is deployed in a production environment (Forsgren et al., 2017). It is thus vital to enterprises to employ strategies to quickly create production-ready software applications.

In the quest to compete in a fast-paced business environment, the complexity of maintaining and enhancing traditional SOA applications, has presented significant challenges to many enterprises (Zimmermann, 2015). These challenges have led to the emergence of a new architectural style to implementing SOA called the microservices architecture (Dragoni et al., 2017, Zimmermann, 2017). The microservices architecture uses a collection of small, loosely coupled software components called microservices that collaborate to automate business functionality, and can be developed within fast software release cycles (Nadareishvili et al., 2016). The microservices architecture promises to provide agility by allowing each microservice to be quickly built, modified, tested and deployed in isolation. The adoption of microservices is thus in line with the emergence of trends that aim to frequently and consistently deliver high-quality working software with minimum project overhead (Bossert, 2016). The success of companies such as Amazon (Bernstein, 2015), Netflix (Ravichandran, Taylor & Waterhouse, 2016), SoundCloud (Baresi, 2017), Facebook (Feitelson, Frachtenberg & Beck, 2013), Google (Kim, 2014) and several others can be attributed to the adoption of the *microservices architecture.*

This chapter presents the state-of-the-art in microservices and microservices architecture. An understanding of the microservices architecture is essential to answer research question RQ1 formulated Chapter 1. First, in Section 3.1 the chapter briefly discusses the various trends in continuous software delivery that have contributed to the adoption of microservices architecture as they are a precursor of a microservices architecture. Section 3.2 introduces and discuss microservices and microservices architecture, including principles and concepts of the architecture. Section 3.3 discuss the collaboration of microservice to automate a business task. Section 3.4 discuss how the location of a microservice instance is identified by other collaborating microservices at runtime. Next, in Section 3.5, the deployment strategies of microservices are discussed. In Section 3.6 an example of collaborating microservices is introduced to demonstrate the concepts of a microservices architecture. The classification of microservices in the microservices architecture is then discussed in Section 3.7. The security challenges of microservices and a conclusion then follow in section 3.8 and 3.9 respectively.

3.1 Trends in continuous software delivery

The concepts of *agile methodology, continuous integration, continuous delivery, and continuous deployment* have significantly changed the way in which online business is enabled (Bossert, 2016). An understanding of these concepts is essential to understanding microservices architecture.

Agile software development is a methodology based on iterative development, where requirements and software evolve as self-organizing cross-functional teams collaborate (Schmidt, 2016). The benefit of agile development is that early feedback is provided on the status of functionality being developed.

Continuous integration is a practice of frequently integrating new software changes into an existing code repository in a manner that ensures that each commit of software source code into the repository results in the compilation and testing of software. Any arising errors can be noticed and corrected immediately (Hilton et al., 2016).

Closely related to *continuous integration* is the concept of continuous delivery. *Continuous delivery* is an extension of continuous integration that makes sure that new software changes are reliably released quickly and sustainably at any time (Loukides, 2012). Continuous delivery requires automating testing and an automated release process that ensure that software changes are deployed any time. *Continuous deployment* refers to the frequent release of software into a production environment.

Over the past few years, a new trend called *DevOps* has emerged that aims to unify agile software development, continuous integration, continuous delivery, continuous deployment with software operations (Bass, Weber & Zhu, 2015). DevOps aims to shorten software development cycles, increase the frequency of deployments and create more dependable software releases that are closely aligned with business objectives (Fowler, 2013). The goals of DevOps are achieved using automation at all steps of software development from integration, testing, releasing software to production and also the management of servers (Davis & Daniels, 2015). Figure 3.1 below shows the relationships between these concepts.

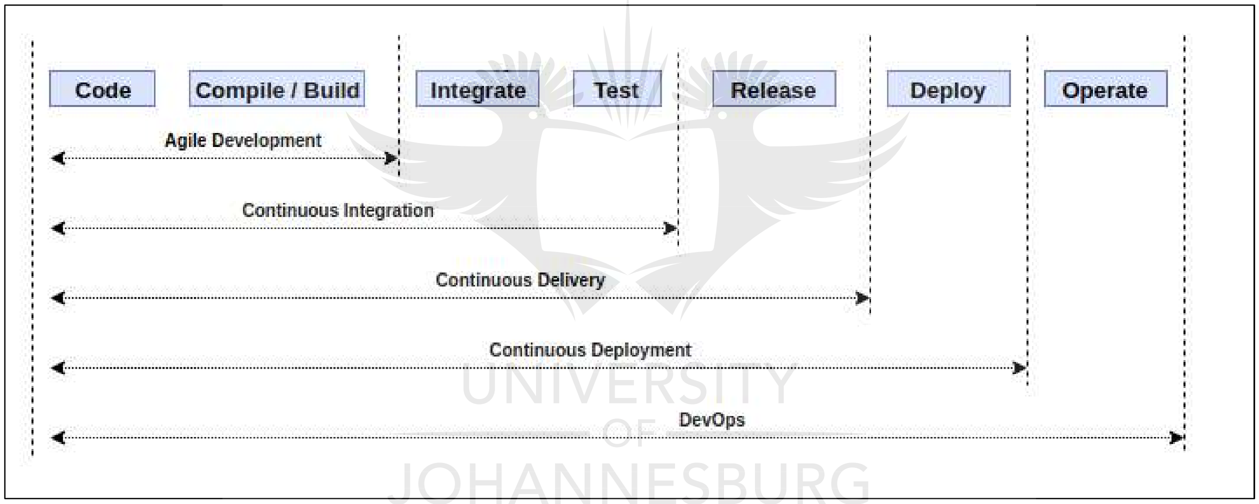


Figure 3.1. The relationship between software development methods

A common set of DevOps and continuous delivery ideologies at companies such as Amazon (Bernstein, 2015), Netflix (Ravichandran, Taylor & Waterhouse, 2016), SoundCloud (Baresi, 2017), Facebook (Feitelson, Frachtenberg & Beck, 2013), Google (Kim, 2014) and several others has led to the adoption of a new architectural style called a *microservices architecture.* Microservices architecture is seen as a natural fit to enable continuous delivery and has become a prelude of a new form of concrete implementation of SOA (Kravchuk et al., 2017).

The next section describes *microservices* and *microservices architecture.*

3.2 Microservices architecture

Microservice architectures aim to overcome the shortcomings of traditional SOA architectures, also called monolithic architectures, where all of the application's logic and data are managed in one deployable unit. To be able make a distinction between these architectures, this section defines and discusses a microservice and the microservices architecture respectively. To illustrate concepts, the PickMeUp example is extended for this purpose. Next, a microservice is defined.

3.2.1 A microservice

A *microservice* is defined as a self-contained, autonomous, lightweight unit of logic running in its own process (Nadareishvili et al., 2016). Microservices communicate using lightweight mechanisms over hypertext transfer protocol (Dragoni et al., 2017), using the RESTful architectural style as a means of communication. A *microservice* provides a narrowly-focused standardized application programming interfaces to its consumers. Microservices have the following characteristics:

- Microservices is a modularization concept (Krivic et al., 2017) where a large application is decomposed into small microservices that communicate using standardized interfaces.
- Each microservice should be designed so that it fulfills only one task and performs the assigned task well (Daya et al., 2016).
- Microservices can be implemented in different technology (Dragoni et al., 2016). There is no restriction on the technology or programming language at hand, as long as the microservice presents a standardized interface for communication. The technology suitable to the work at hand is adopted.
- Microservices are deployed independently of other microservices (Thönes, 2015). Each microservices is a self-contained process that can run on its own. Changes to one microservices can be taken into production, independent of changes made to other microservices. This makes it easy to roll-back features that fail after new deployments are made. The automated deployment of a microservice should preferably be applied (Thönes, 2015, Zimmermann, 2016).
- Microservices should easily be replaced by other microservices offering the same communication interface (Le et al., 2015) to reduce the overall risks of incorrect decisions made at development time.

- Microservices should be designed so that that they can work together to perform a task.

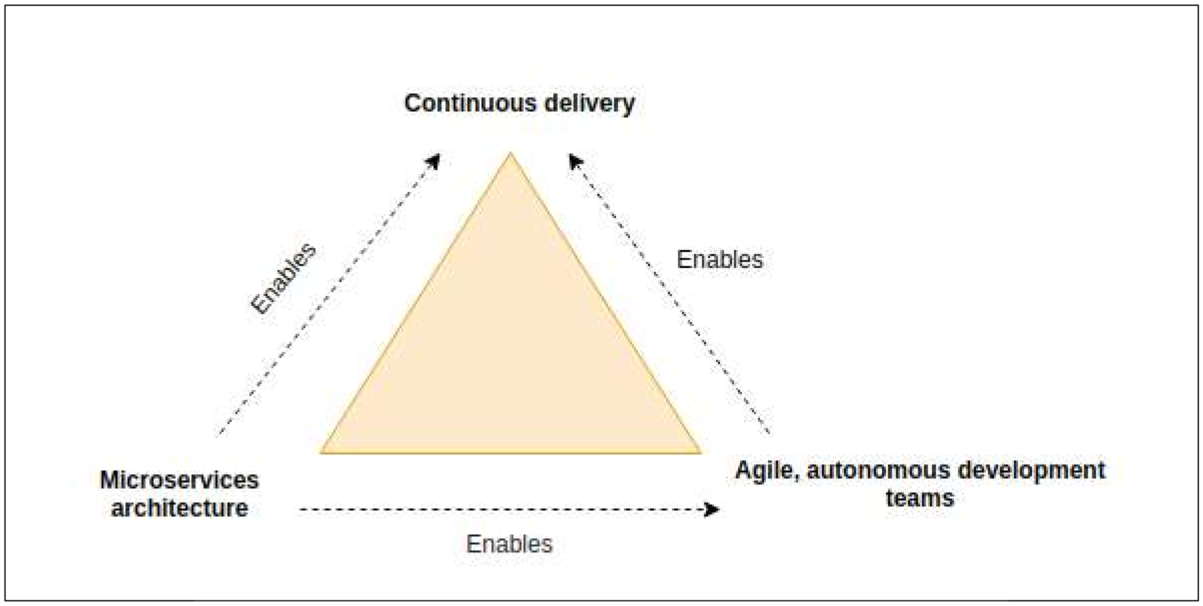
The adoption of microservices brings many benefits such as:

- *Agility* – due to shortened build, test, and deployment cycles (Lawton, 2015, Killalea, 2016). Each microservice can incorporate the flexibility needed to employ microservice's specific needs for replication, persistence, monitoring, and security.
- *Reliability* – due to the fact that a fault with one microservice only affects that microservice and its consumers, unlike a single-tier application were a failure affects the entire application (Balalaie, Heydarnoori & Jamshidi, 2015).
- *Availability* – due to minimal downtime required when deploying a new version of a microservice (Jose & Shettar, 2017). Only the microservice being deployed is impacted, and the entire application that uses microservices does not require a full restart of the whole application.
- *Modifiability* – due to the flexibility to adopt or consume new frameworks, libraries, data sources, and other resources. Microservices tend to be easier to work with and to understand (Le et al., 2015).
- *Management* – due to the use of agile methodology, where the development effort is divided across teams that are smaller and work more independently (Newman, 2015, Zúñiga-Prieto et al., 2016).

3.2.2 Microservice architecture

The *microservices architectural style* is an approach that structures an application as a set of loosely coupled collaborating microservices. There are no set of rules when choosing between various frameworks or protocols to use in a *microservices architecture*. However, the protocol should be lightweight, keeping in mind that the microservices architecture relies heavily on messaging between collaborating microservices. Using this architecture style, an enterprise can structure development teams as a collection of small autonomous teams, usually at most nine members, who focus on one or more microservices (Lalsing, Kishnah & Pudaruth, 2012).

Figure 3.2 shows the relationship between microservices architecture, continuous delivery, and small, agile autonomous development teams.


Continuous delivery
Enables
Enables
Microservices architecture
Agile, autonomous development teams
Enables
Figure 3.2. Microservices architecture, continuous delivery, and autonomous teams

Next, the PickMeUp example is extended to illustrate the various principles and concepts behind the microservices architecture.

3.2.3 PickMeUp microservices example

The PickMeUp SOA application is now decomposed into a set of smaller, collaborating microservices to illustrate the principles and practice of microservices architecture. The benefits of microservices architecture increase when the functional scope of each microservice is carefully considered (Dragoni et al. 2017). The recommendation is that a microservice should correspond to an organization's business capabilities (Newman 2015, Balalaie, Heydarnoori & Jamshidi 2015, Dragoni et al. 2017). A business capability is defined as an activity that a business does, to generate value (Sandkuhl & Söderström 2016). For example, in the PickMeUp SOA application business model discussed before, the management of passenger information is identified as an example of a business capability. The following are useful guidelines that define the functional scope of microservices:

- The Single Responsibility Principle (SRP) establishes the responsibility of microservice to be limited only to a single part of the business functionality of the application (Rahman & Gao 2015, Killalea 2016). For example, in the PickMeUp application, a microservice to maintain driver details should only focus on that use case. When the SRP is applied, the microservice will change when its business functionality changes. SRP ensure that the functionality of each microservice is isolated from another microservice and changes on one microservice does not require other microservices to change.

- The Common Closure Principle (CCP) states that software components that change for the same reason should be grouped in the same package (Albattah & Melton 2014). Any business functionality that is likely to change for the same purpose or is tightly coupled should be in the same microservice. This can ensure that any change in business requirements impacts only a single microservice.

The microservices architecture decomposes an application into collaborating microservices. In the PickMeUp application, the first step is to identify business capabilities. Then, the functional context of each microservices is determined using SRP and CCP. Figure 3.3 below shows an example of how the operational context of microservices is defined from the business capabilities. Three capabilities namely *Passenger Management*, *Driver Management*, and *Trip Management* are used as an illustration.

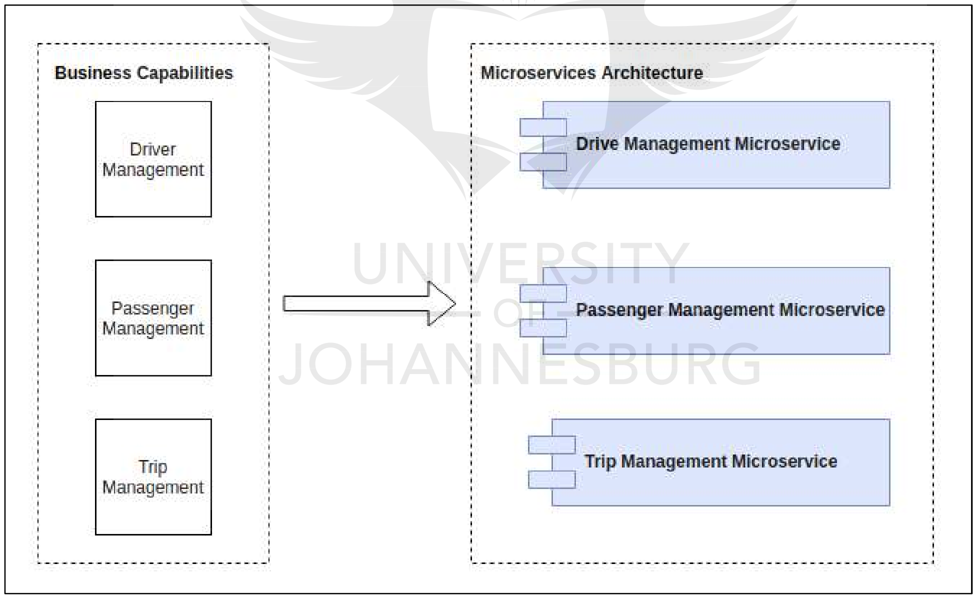

Business Capabilities
Driver Management
Passenger Management
Trip Management
Microservices Architecture
Drive Management Microservice
Passenger Management Microservice
Trip Management Microservice

Figure 3.3. Mapping business capabilities to microservices

Once the functional scope is identified, the technology that is suitable for each microservices at hand is chosen as they each can be developed using a different technology stack. The essential requirement for the microservices architecture is that independent microservices collaborate to automate a business functionality. When microservices work together to fulfill a business task, a *microservices composition* is created. The next section discusses how a *microservices composition* is formed.

3.3 Microservices compositions

In a microservice architecture, *choreography* is preferred when creating a microservices composition, unlike traditional SOA, were orchestration is used. In choreography, there is no central microservice called a *composition controller* that controls communication with other microservices (Butzin, Golatowski & Timmermann, 2016). Microservices in choreography communicate using point-to-point exchanges or by listening to events on their environment (Sheng et al., 2014). The inter-communication mechanisms can either be synchronous or asynchronous, discussed next.

3.3.1 Synchronous communication

Synchronous communication is a point-to-point style of communication were microservices communicate directly with each other in a blocking way. For each request that is sent, the calling microservice waits for a response. The entire message routing logic resides on each microservices.

Figure 3.4 shows synchronous point-to-point communication between the *Trip Management Microservice* and the *Driver Management Microservice*. The *Trip Management Microservice* invokes the *Driver Management Microservice* by sending a request using the REST architectural style and waits until a response is received from the later.

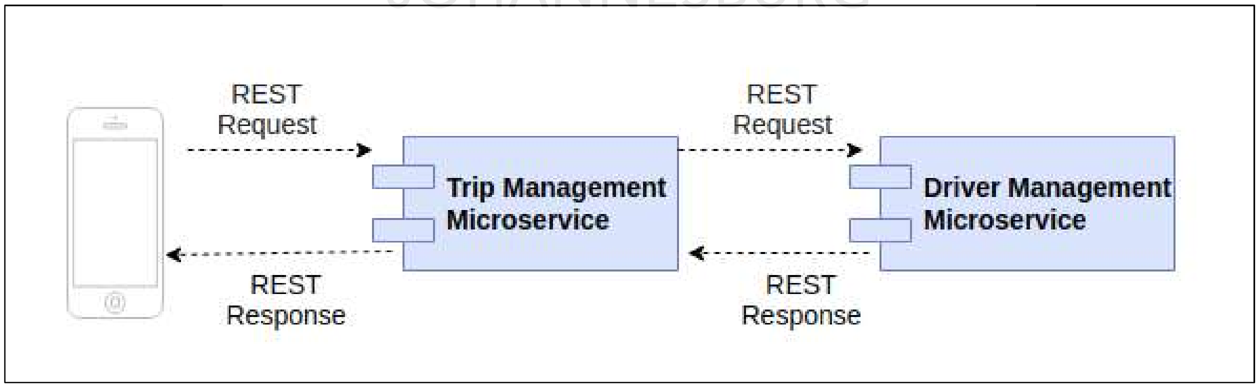


Figure 3.4. Synchronous microservices communication

The synchronous model of communication works well for relatively simple microservices compositions. As the number of microservices increases in the microservices composition, synchronous communication becomes overwhelmingly complex. The disadvantages of this model of communication is:

- When microservices synchronously invoke one another, there is the possibility that one microservice may be unavailable or exhibit high latency (Newman, 2015). The failure of one microservice can potentially cascade to other microservices throughout the microservices composition.
- Non-functional requirements such as monitoring have to be implemented at each microservice within the composition to mitigate against failures resulting from a single microservice (He & Yang, 2017).
- Many microservices typically run in a virtualized environment where the number of microservices instances, and their locations change dynamically. Consequently, mechanisms are required to enable each microservice to make requests to other dynamically changing sets of ephemeral microservices instances (Rotter et al., 2017).
- Microservices may use a diverse set of protocols, some of which might not be web-friendly (Richardson, 2016). This may require each microservices to be equipped with the logic to transform messages.

The challenges of implementing a synchronous model of communication in a microservices composition make asynchronous communication a more suitable approach. Asynchronous communication is discussed next.

3.3.2 Asynchronous communication

In asynchronous communication, several channels are used to exchange messages. Microservices are connected to a message bus and subscribe to channels of interest (Dragoni et al., 2017). Any number of microservices can send messages to a channel. Similarly, any number of microservices can receive messages from a channel. There are two kinds of message channels, namely *point-to-point* and *publish-subscribe*.

- A *point-to-point* channel delivers a message to exactly one microservice that is reading from the channel.

- A *publish-subscribe* channel delivers each message to all microservices that subscribe to the channel. Microservices use *publish-subscribe* channels for the one-to-many interaction styles were each request is processed by multiple service instances.

Figure 3.5 shows an asynchronous model of communication using the microservices from the on-demand taxi application example. The *Trip Management Microservice* publishes a message to a channel that is of interest to both the *Passenger Management Microservice* and the *Driver Management Microservice*.

Here, the asynchronous model of communication has the following advantages:

- Easy to add new microservices to the microservices composition as a new microservice is added by connecting the microservice to the message bus and ensuring that other microservice emit the events required by the new microservice (Newman, 2015).
- Microservices are decoupled from each other in the microservices composition making them independent of each other (Richter et al., 2017).

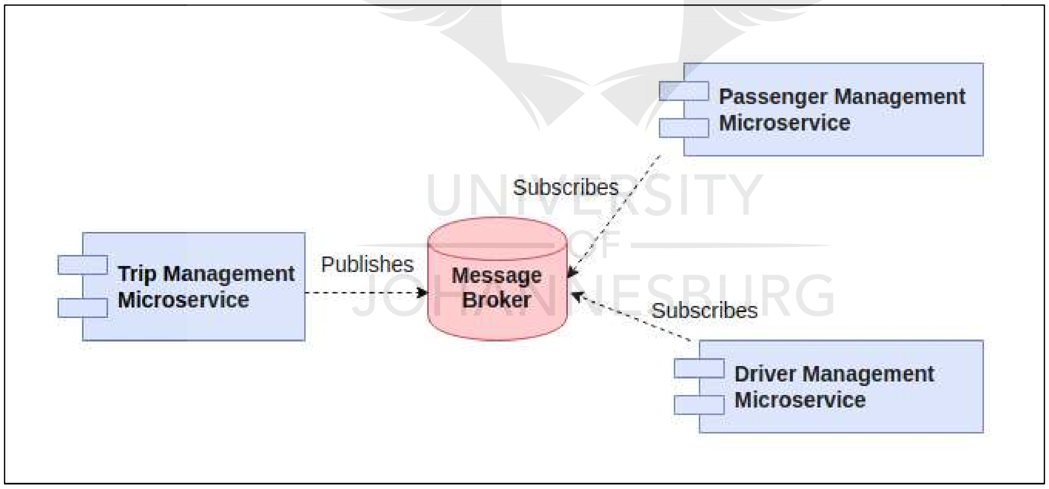

Passenger Management Microservice
Subscribes
Trip Management Microservice
Publishes
Message Broker
Subscribes
Driver Management Microservice

Figure 3.5. Asynchronous communication between microservices

As microservices expose a fine-grained application programming interface (API), it can be a challenge when there is a mismatch between the needs of various external clients. For example, the desktop browser client typically can consume an API that provides more elaborate details than mobile clients.

Next, the use of an application programming interface gateway to address different needs of microservices clients is discussed.

3.3.3 Application programming interface gateway

The API gateway is an essential component of the microservices architecture. The API gateway acts as a lightweight entry point for a diverse set of external clients (Montesi & Weber, 2016). Zuul (Netflix, 2013) from Netflix is an example of an implementation of the API gateway pattern (Macero, 2017). Figure 3.6 shows an example of an API gateway that exposes the functionality of the *Trip Management Microservice* and the *Payments Microservice* as light-weight API to mobile clients.

The API gateway provides the following benefits.

- It gives the ability to provide a different application programming interface that is suitable to the needs to each client (Montesi & Weber, 2016).
- The gateway can be used to provides lightweight message routing and transformation according to the requirements of each microservice (Alpers et al., 2015).
- The gateway provides a central place to apply non-functional requirements such as security and monitoring (Balalaie et al., 2015).
- The gateway makes microservices to become more lightweight as all the non-functional requirements are implemented at the gateway (Montesi & Weber, 2016).

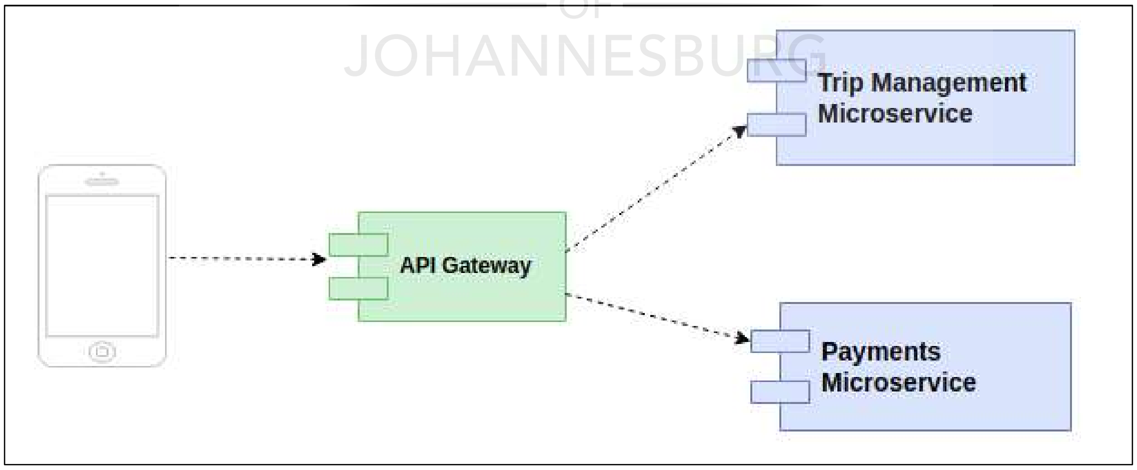


Figure 3.6. Microservices API gateway

The gateway provides the following benefits.

- It gives the ability to provide a different application programming interface that is suitable to the needs to each client (Montesi & Weber, 2016).
- The gateway can be used to provides lightweight message routing and transformation according to the requirements of each microservice (Alpers et al., 2015).
- The gateway provides a central place to apply non-functional capabilities such as security and monitoring (Balalaie et al., 2015).
- The gateway makes microservices to become even more lightweight as all the non-functional requirements are implemented at the gateway (Montesi & Weber, 2016).

In the microservices architecture, microservices discovery is an essential aspect of the microservices architecture (Balalaie, Heydarnoori & Jamshidi, 2015). Discovering microservices becomes essential due to the deployment of microservices in virtualized environments. When microservices are deployed in virtualized environments, strategies to locate microservices are essential because the network location of microservices are assigned dynamically (Rotter et al., 2017). The next section discusses how microservices instances are discovered in the architecture.

3.4 Microservices discovery

Microservices discovery utilizes a microservices registry (Montesi & Weber, 2016). A microservices registry is a database of microservices, their instances, and their locations. Microservices cases are registered with the microservices registry when the microservice starts up and de-registered when the microservice shuts down. An example of a microservices registry is Eureka developed by Netflix (Netflix, 2012). The following approaches are used to locate an instance of a microservice:

- *Client-side discovery.* In this approach, clients of a microservices directly query the microservices registry to find the locations of a microservice instance (Montesi & Weber, 2016). The limitation of this approach is that it couples the microservice client to the microservice registry. Figure 3.7 shows an example of client-side discovery. At runtime, the *Trip Management Microservice* queries the registry for the location of the *Driver Management Microservice*.

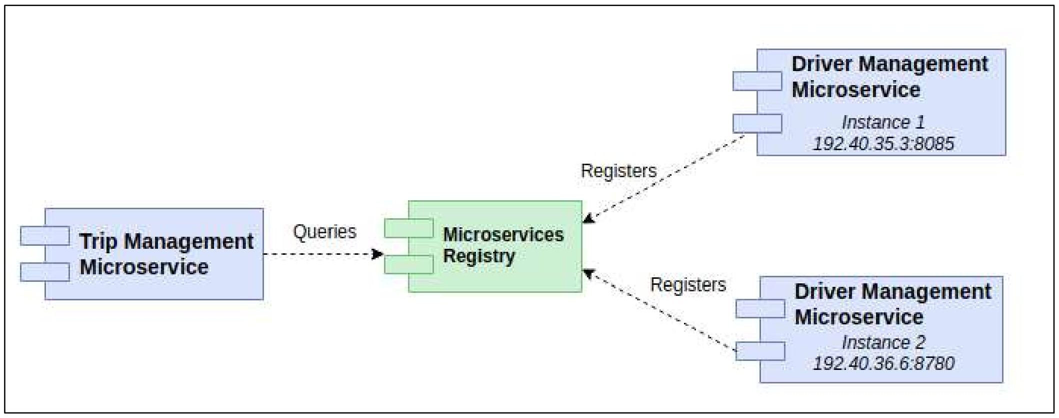


Figure 3.7. Microservices client discovery

- *Server-side discovery.* When making a request to a microservice, the client makes a request via a router or a load balancer located at a known static location (Montesi & Weber, 2016). The router or load balancer then queries a service registry, which might be built into the router. Once the router or load balancer discovers the location of the microservice instance from the registry, it then forwards the request to microservices instances. AWS Elastic Load Balancer (Guide, 2010) is an example of a server-side discovery router. Figure 3.8 shows microservices using server-side discovery. The *Trip Management Microservice* forwards a request to the router, that is responsible for locating and delivering the request to microservices instances.

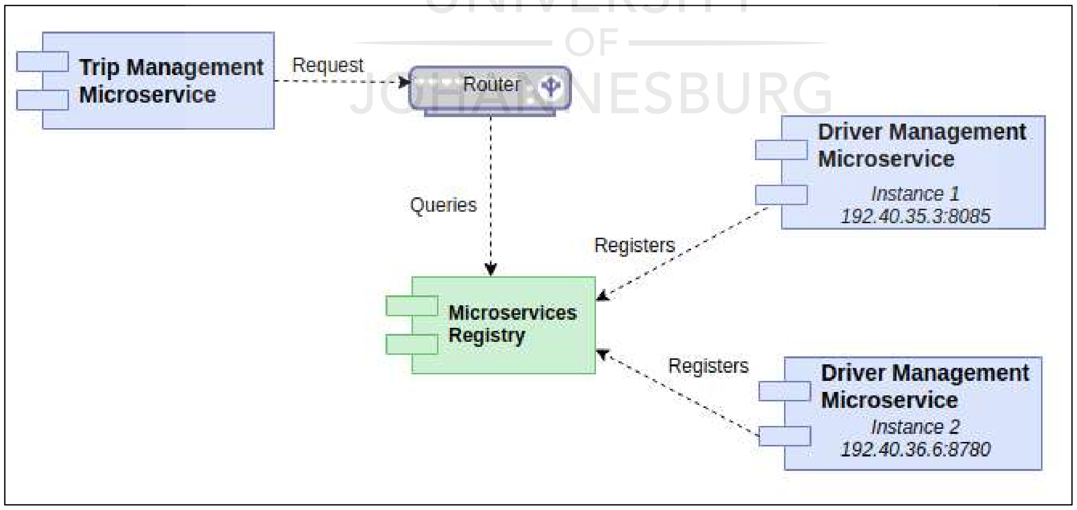


Figure 3.8. Server-side microservices discovery

3.5 Microservices deployment strategies

Various deployment strategies can be used to deploy microservices. These are:

- Deploying multiple microservices instances per host (Dragoni et al., 2015). The host may be a physical or virtual machine. Each microservice instance is deployed as a process on the host or multiple service instances are deployed in the same virtual machine (Newman, 2015). The limitation of this approach is the risk of conflicting resource requirements for each microservices deployed on the same host. It may also be difficult to isolate each microservice instance or to monitor and limit the resources consumed by a single microservice instance.
- Deploying each microservice instance on its own host (Johansson, 2017). The benefits of this approach are that microservices instances are isolated from one another. There is no possibility of conflicting resource requirements for microservices. Monitoring, managing, and redeploying of a microservice instance is simplified. The drawbacks are that there is potentially less efficient resource utilization compared to running multiple microservices per host.
- Deploying one microservice instance per container (Jaramillo, Nguyen & Smart, 2016). A container image is a filesystem image consisting of the microservice and libraries required to run the microservice (Merkel, 2014). The microservice is packaged as a container image and deployed as a container. The benefit of this approach is that it is straightforward to scale a microservice by changing the number of container instances. Docker containers (Merkel, 2014) are becoming a common container technology for packaging and deploying services (Anderson, 2015). Each microservice is packaged as a Docker image. Containers are extremely fast to build and start (Merkel, 2014).
- Serverless deployment (McGrath & Brenner, 2017). This uses a deployment infrastructure provided by public cloud providers. Code for a microservice is packaged and uploaded into the deployment infrastructure provided by the cloud providers. The providers hide the concept of servers, physical or virtual hosts, or containers. Examples include AWS Lambda (Sbarski & Kroonenburg, 2017), Google Cloud Functions (Wagner & Sood, 2016), Azure Functions (Baldini et al., 2017). The infrastructure runs the microservices, and the infrastructure providers charge based on resources consumed. The serverless deployment infrastructure automatically scales microservices to handle the load. The benefits of using serverless deployment are that it eliminates time to spend low-level managing infrastructure by the development team. The drawback is that the deployment environment typically may have far more constraints on

supported languages, for example, Amazon Web Services AWS Lambda (McGrath & Brenner, 2017). This approach is also not suitable for long-running stateful applications (Baresi, Mendonça & Garriga, 2017).

The next section summaries the various concepts of the microservices architecture discussed above using the PickMeUp example.

3.6 Microservices composition example

The various architectural components of the microservices composition for the PickMeUp application are shown in Figure 3.9 namely the API gateway, a service registry, and message broker. The deployment strategy adopted for the application is to deploy components of the application on separate Docker containers that run on a single host.

Figure 3.9 above shows a set of collaboration microservices that form a microservices composition. Access to the composition is done via the API gateway. The gateway locates the instance of a trip management microservices using the service registry. The trip management microservices communicate either directly by calling other microservices' REST interface or by sending a message to the message broker. Each microservice is deployed in its Docker container.

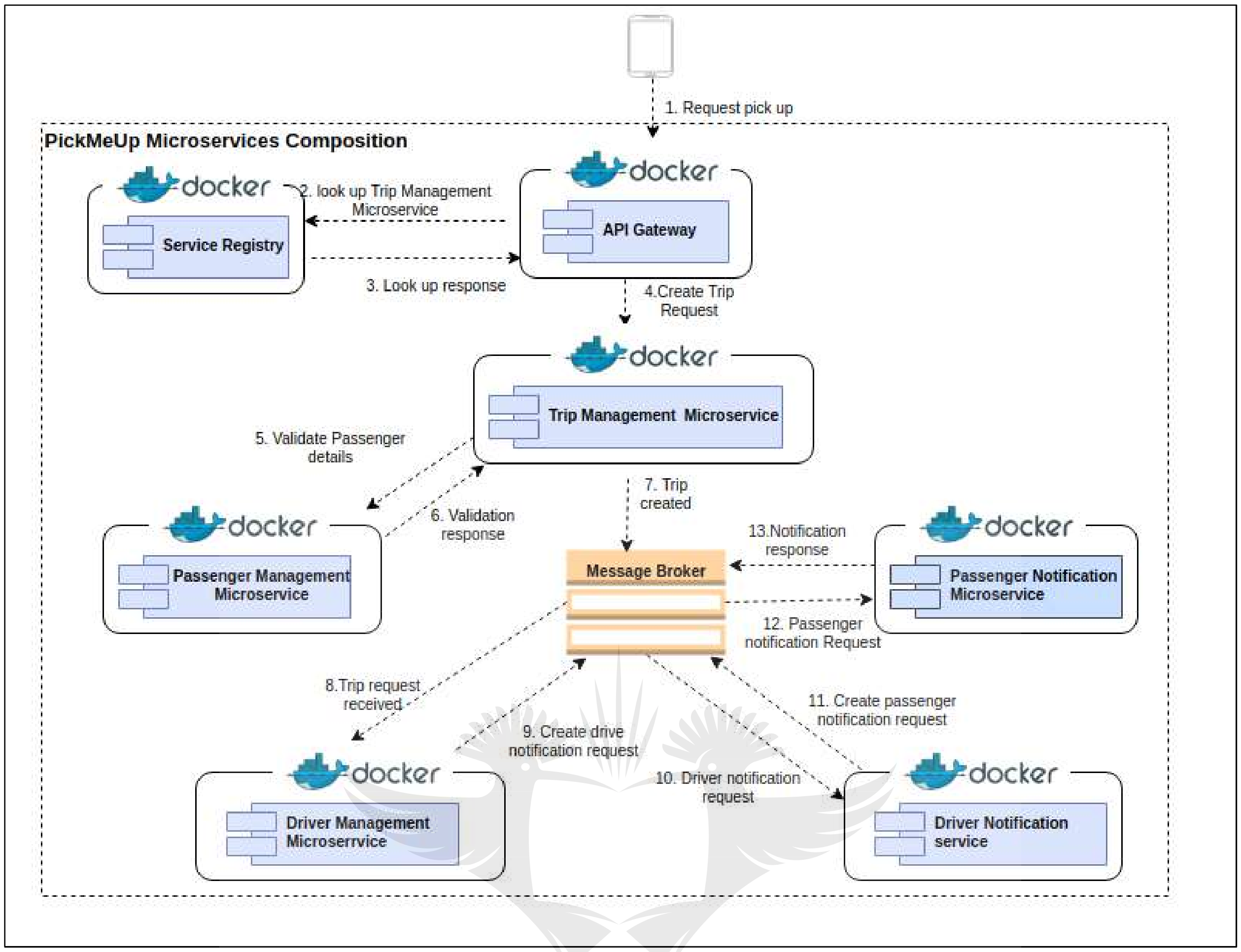

1. Request pick up
PickMeUp Microservices Composition
docker
Service Registry
2. look up Trip Management Microservice
3. Look up response
API Gateway
4.Create Trip Request
Trip Management Microservice
5. Validate Passenger details
6. Validation response
Passenger Management Microservice
7. Trip created
Message Broker
13.Notification response
12. Passenger notification Request
Passenger Notification Microservice
8.Trip request received
9. Create drive notification request
10. Driver notification request
11. Create passenger notification request
Driver Management Microserrvice
Driver Notification service

Figure 3.9. PickMeUp microservices composition

Another important aspect of the microservice architecture is the classification or taxonomy of microservices, discussed next.

3.7 Microservices architecture taxonomy

The microservices architecture has a limited microservice taxonomy. Microservices are classified into *functional microservices* and *infrastructure microservices* (Richards, 2015).

- *Functional microservices* support business requirements. They are accessed using an application programming interface (API) which acts as a microservice facade. The API serves as an abstracting layer so that changes can be made to the service without affecting the consumer.
- *Infrastructure microservices* supports non-functional tasks such as auditing, logging, and monitoring. They are generally not accessible externally.

Figure 3.10 below shows the taxonomy of microservices. A functional microservices is accessible to the external client through an application programming interface. The functional microservice delegates all its non-functional tasks to the infrastructure microservice.

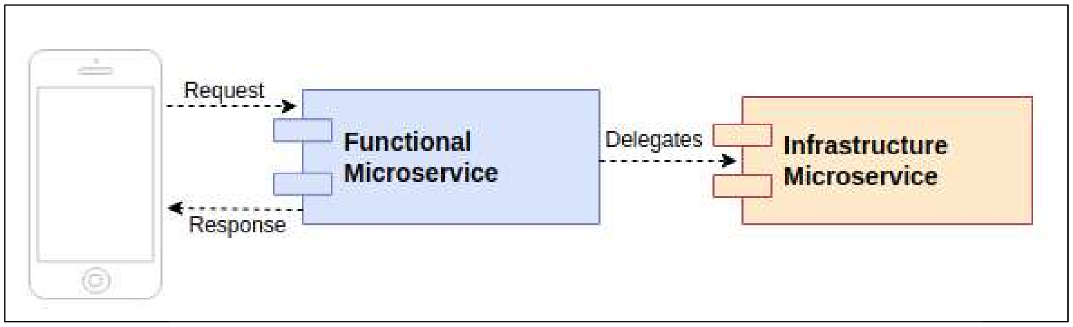


Figure 3.10 Microservices service taxonomy

The limited microservice taxonomy makes microservices ownership less complicated. An application development team may own the functional microservices and the infrastructure microservices. This allows development teams to be broken down into smaller independent teams whose work is integrated as it is delivered. There is less coordination among teams to provide a microservice. This fosters complete ownership by self-contained teams making development, testing, and maintenance less complicated. Figure 3.11 depicts microservices ownership model in the microservices architecture.

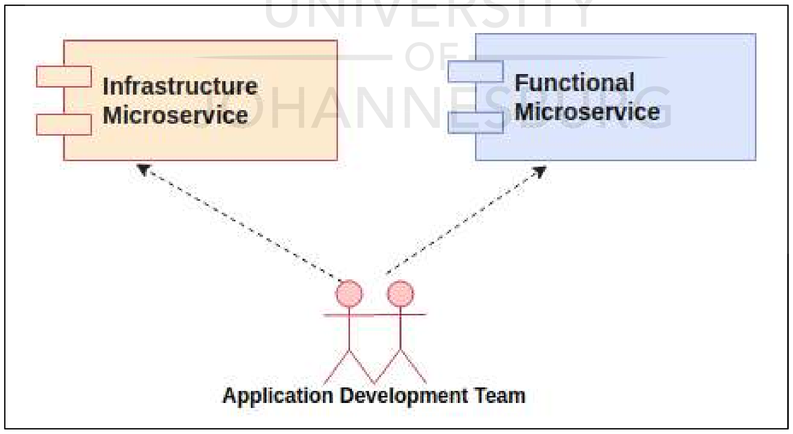


Figure 3.11. The microservices service ownership model

3.8 Microservices architecture security challenges

Despite the undeniable success of microservices architecture, the biggest challenge is security. The adoption of microservices architecture as part of DevOps practices introduces complications when implementing security controls. As development teams continue to deliver software in short and agile sprints cycles, usually one to two weeks in length, often little attention is given to the security of the application.

The preferred models of deploying microservices provide an attacker with increased options to attack microservices compositions. In most instances, the myriad of distributed microservices are often designed to trust each other completely. A compromise of a single microservice could bring down the entire application.

Furthermore, many of the most popular tools used for ensuring continuous integration, continuous delivery and continuous deployment and DevOps are often new to the market or are open-sourced. The relative immaturity leads to concerns about the degree to which secure development standards are being adhered to. Most of the security challenges arise from the way microservices are deployed.

3.9 Conclusion

The past few years have seen the emergence of agile methodology, continuous integration, continuous delivery, and continuous deployment and DevOps whose aim is to shorten software development cycles and increase the frequency of software deployment. The objective of these methodologies is to quickly build potentially shippable software increments and bringing changes to production as soon as possible. A common set of DevOps ideologies at various companies has to lead to the adoption of a new architectural style called microservice architecture that decomposes applications into small units of logic called microservices.

Microservices communicate synchronously or asynchronously to fulfill a business task using lightweight protocols. Collaborating microservices create a microservices composition. Adoption of the microservices architecture has been one of the DevOps success stories. This chapter has presented the various aspects that captures the fundamental understanding of microservices architecture. The benefits of adopting this architectural style have also been introduced.

Providing secure and reliable microservices-based applications is increasingly needed to ensure successful adopting of microservices architecture. The next chapter discusses the fundamental concepts of web services security which provides the basic building blocks to understanding SOA and microservices security. The chapter also shows how the current security practices in SOA falls short when applied to the new microservices architecture.

Chapter 4

Security of Web Services

4.0 Introduction

The development of secure microservices applications is not a problem that has been solved. To date, little research focuses on microservice security. However, the core security principles that apply to SOAP-based, and RESTful web services hold for microservices as well. There is a large body of work on security protocols and security best practices for web services that can be used when building microservices applications.

SOA security is a topic that has been extensively discussed in literature (Buecker et al., 2008, Kanneganti & Chodavarapu, 2008, Shashwat, Kumar & Chanana, 2017). When distributed SOA applications were developed, secure silo-based application logic had to be made available to external partners, leading to a major change in how services need to be protected. SOA security comprises of general security standards as well as web services security standards that are generally XML-based. Security requirements of web services are specified in a security policy document, and referenced within the WSDL. WS-Security (Nadalin et al., 2006) specifies the way integrity and confidentiality can be enforced, and security tokens used for authentication. Middleware is used to enforce distributed security via components such as interceptors (Shah & Patel, 2008).

In contrast, the RESTful style of web services does not provide any formal guidance on how security mechanisms should be applied and leaves their implementation to the discretion of software engineers. Such services mainly rely on ad-hoc security mechanisms or transport layer security. With microservices, security becomes more of a challenge because no middleware component is available to manage security-based functionality. Instead, each microservice is required to manage security on its own, or in other cases, the API gateway is given the responsibility of managing the security of the application.

In order to provide a foundation for this research, this chapter provides a background to the security of both traditional SOA implementations and microservices architecture implementations. Section 4.1 defines security concepts. Section 4.2 identifies general information security services that are required to meet the security requirements of both the traditional SOA-based and microservices-based systems. Section 4.3 discusses web services security vulnerabilities and section 4.4 then introduce the web services security model. Section 4.5 discusses the implementation of the web service security model in SOAP-based web services. Section 4.6 discusses the implementation of the web service security model for RESTful web services. Section 4.7 then identify the new security challenges of microservices. A conclusion then follows in section 4.8.

4.1 Information security concepts

The information security of both traditional SOA and microservices architecture implementations can be defined as the degree to which malicious harm to assets of the application is prevented, reduced, and adequately responded to. The objective of information security is to protect valuable or sensitive information while making the information readily available to the users of the application (Kissel, 2013). Figure 4.1 below adapted from Firesmith (2004) shows the relationships between various information security concepts defined above.

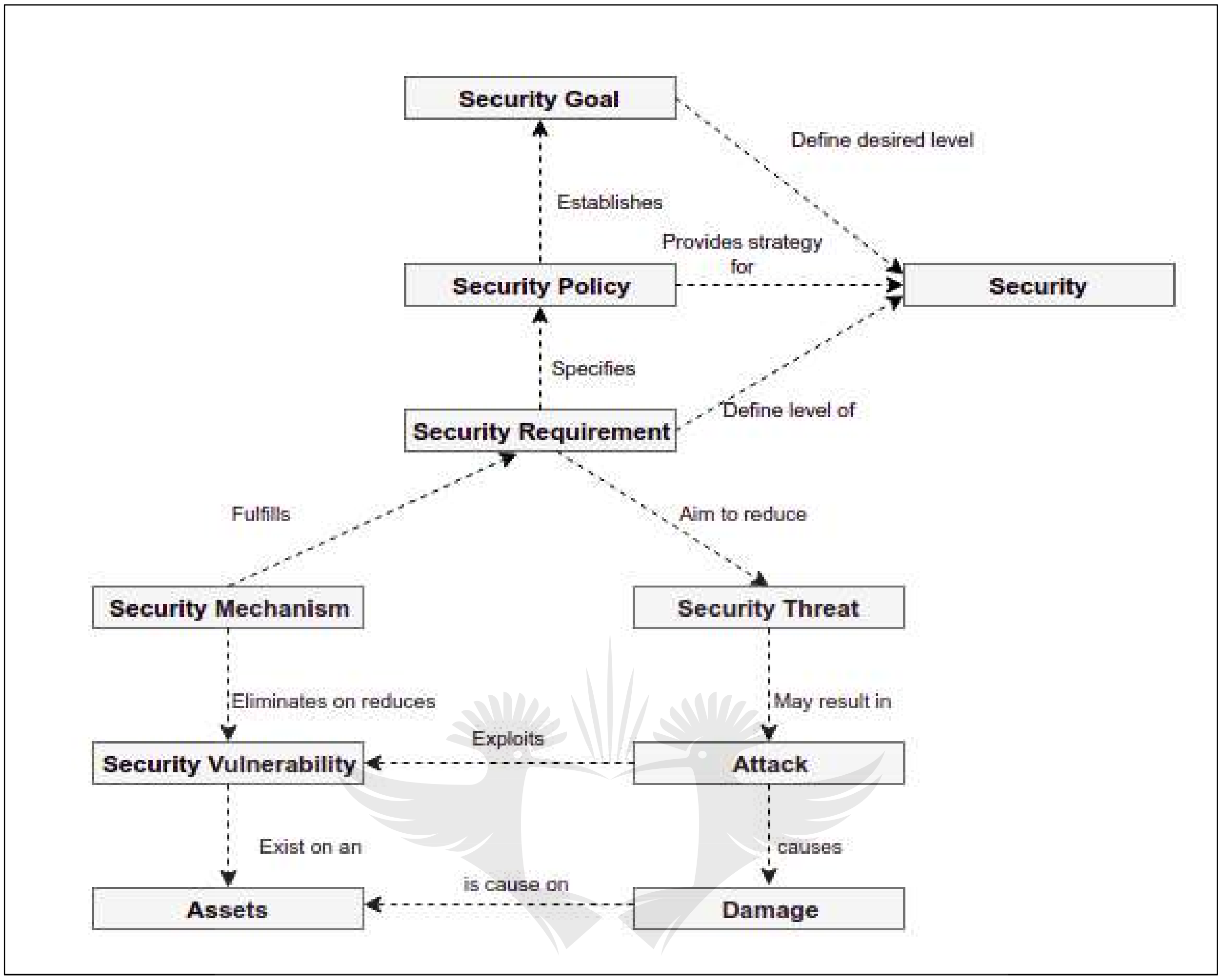


Figure 4.1 Security concepts relationships

The vital information security concepts are:

- *Asset* - anything that has value to an organization's operations and continuity (Kissel 2013, Priya & Arya, 2016) such as services, servers and information.
- *Attack* - an unauthorized attempt to cause harm to assets (Kissel, 2013).
- *Attacker*- an agent that initiate an attack to cause harm to assets, by disrupting normal operations or stealing information, using attack methods, tools, and techniques (Priya & Arya, 2016).
- *Vulnerability* - a weakness in the system's requirements, system's design, system's implementation or operation that an attacker can exploit to achieve a malicious motive (Kissel, 2013). A dictionary of all publicly known information security vulnerabilities or exposures are documented by the Common Vulnerability Exposure (CVE) project (Mitre 2017).

- *Attack surface* - the sum of vulnerabilities in a given system that is accessible to an attacker (Giarratano, Guise & Bodin, 2017).
- *Threat* - any circumstance or event that creates a possible danger that might be exploited by an attacker to breach the security of a system (Bertino et al., 2009, Kissel, 2013). A threat exists typically when an entry point into the system provides access to an asset (Priya & Arya, 2016).
- *Security goal* - a desirable ability of a system to resist a specific category of threats (Cherdantseva & Hilton, 2013).
- *Security policy* - an aggregation of directives, regulations, rules, and practices that prescribes how assets are protected, and how information is distributed in a secure manner (Ross, McEvilley & Oren, 2016). A security policy represents a set of security constraints that must be enforced to assure secure access to assets.
- *Security mechanism* - a method, tool, or procedure for enforcing a security policy. It is a countermeasure that helps reduces one or more security vulnerabilities (Ross, McEvilley & Oren 2016).
- *Security requirement* - the functional, assurance, and strength characteristics of a protection mechanism (Kissel, 2013). It is a quality of service requirement that specifies a required level of security using system-specific criteria (Penzenstadler et al., 2014).

Both traditional SOA-based applications and those based on the microservices architecture need to be able to handle traditional security demands of protecting information and ensuring that access is only granted to entities that are permitted. ISO 27002 (ISO, 2013) defines five categories of information security services to meet the global and pervasive security requirements of any given information system including service-oriented applications. At the time of writing ISO 27002 was undergoing revision to cater for more security services. The next section identifies the information security services that are required to ensure secure applications.

4.2 Information security services

Both traditional SOA-based and microservices-based applications need to be able to resist security threats. ISO 27002 defines security services that can provide protection to achieve this goal. The five information security services identified by ISO 27002 are:

- *Authentication* – the information security service that ensures that an entity is identified before access to a resource is granted (ISO, 1989). In web services, a*uthentication* requires that web service must provide proof that its claimed identity is true (Erl, 2008). In general, an entity can prove identity by presenting what they know with a username and password, or what they have for example an authentication token, or what they are with a biometric.
- *Access control* - the information security service that controls what type of access an entity is granted to a resource. The decision to grant access may depend on criteria such as the action the entity wants to perform, the resource on which the action is being requested, and the groups the requester belongs to.
- *Data confidentiality* – the information security service that is concerned with protecting the privacy of the contents of a message. A message is considered to have remained confidential if in its message path no service or agent that is not authorized to do so viewed its contents. Confidentiality can be enforced by defining appropriate access levels for information (ISO, 1989).
- *Data integrity* - the information security service that ensures that information in transit is not tampered with, or any tampering of information is be detected (ISO, 1989). In the context of web services, i*ntegrity* ensures that the state of the message contents remains intact from the time of transmission to the point of delivery (Erl, 2008).
- *Non-repudiation (non-deniability)* – the information security service that ensures that the entity cannot deny creating or modifying the resource after the fact (ISO, 1989).

Figure 4.2 below shows how information security services are positioned to protect the assets of an application (Yamany, Capretz & Allison, 2010). The information security services may be implemented as components to ensure that access to assets is only granted to entities that are permitted.

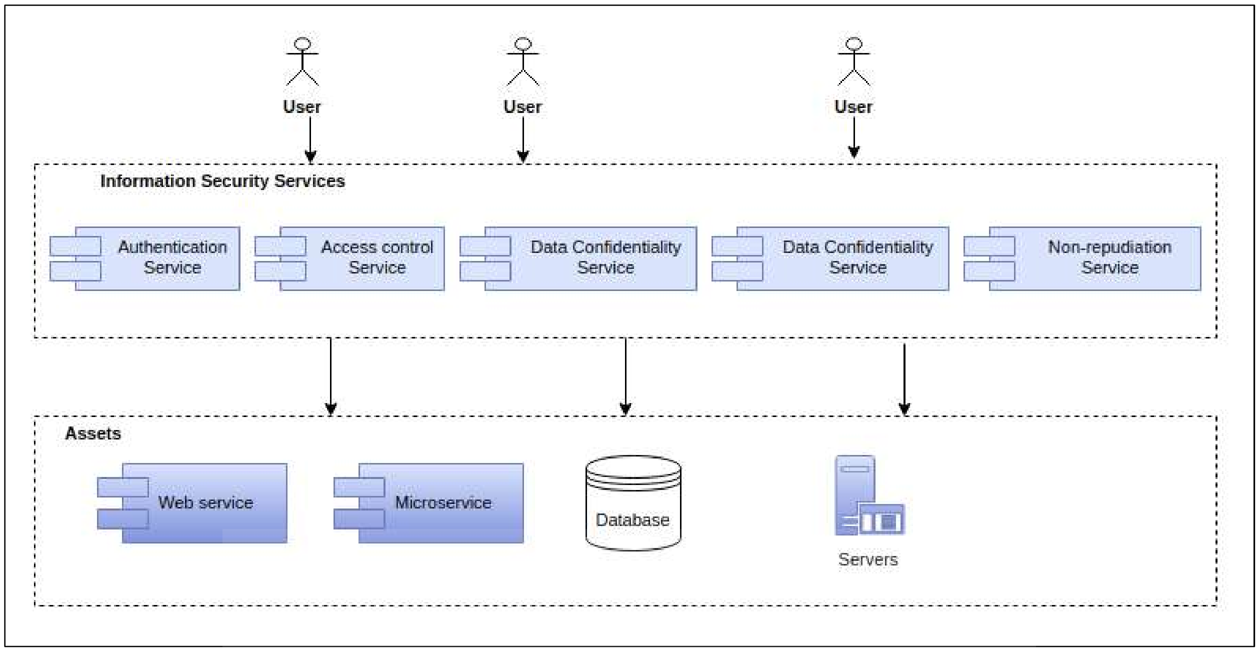


Figure 4.2. Information security services and SOA assets

The challenge of using web services is that they have known security vulnerabilities that an attacker can exploit to compromise an application. The next section briefly identifies the common security threats associated with web services.

The following are common threats that can occur for SOAP-based and REST-based web services that make use of XML messages:

- *Buffer overflows* occur when an attacker crafts an XML message in such a way that the XML message references its elements recursively. This causes a memory overflow when the XML is parsed and may trigger error messages which reveal information about the web service. The server parsing an XML file may repetitively use more resources to parse the file, and this can result in denial of service (DOS) (Chan, Chua & Lee, 2016).
- *XML injections* occur when a server does not validate data correctly. A malicious web service message may be used to create XML data which inserts a parameter into an SQL query and send to the server which then executes the message using the rights of the web service. Another attack is to poison the schema, a file that an XML parser uses to understand the XML's grammar and structure. This allows the XML parser to process malicious web service messages (Chung et al., 2014).
- *Session hijacking* involves gaining unauthorized control of a legal user's valid session state and

use it to gain that particular user's privileges in the application. This is done by intercepting or sniffing web services messages (Chung et al., 2014).

There are numerous more threats that can compromise the confidentiality, integrity, or availability of SOAP-based and REST-based web services (Vorobiev & Han, 2006, Popa, 2015). Ensuring that web services provide authentication, access control, integrity, confidentiality, and non-repudiation is critical for an enterprise and its customers. The next section discusses a generic security web service model that can be used to support information security services, discussed in section 4.2 above.

4.3 Generic web service security model

The web service security model, shown in Figure 4.3, supports, integrates and unifies several popular security models, mechanisms, and technologies, and enables a variety of systems to interoperate securely (Della-Libera et al., 2002). The generic model can apply to both SOAP and REST web services and may thus also apply to microservices-based applications. The web service security model is generic and can be used to support more specific security models. The aim of the security model for web service security is to ensure that:

- A web service, both SOAP and REST, can require that an incoming message provides information that expresses its origin and ownership. The expression of such information is referred to as making a *claim*. A *claim* is a statement about a subject that is used for example to assert the subject's identity or the subject's authorized role (Bigdoli, 2006). Claims can be a name, security key, permission or a capability. If a message arrives without having the required claims, the web service may ignore or reject the message. A set of claims and related information is called a *policy* (Della-Libera et al., 2002).
- A requester, which can be an end user or another web service can send messages with proof of the required claims by associating *security tokens* with the messages. Thus, web services messages both demand a specific action and prove that their sender has the claim to demand the action.
- Another web services can be contacted to provide the required claims. The other web services referred to as *security token services*, may, in turn, require a set of claims (Della-Libera et al., 2002). The *security token services* are used to broker trust between different trust domains by issuing *security tokens*.

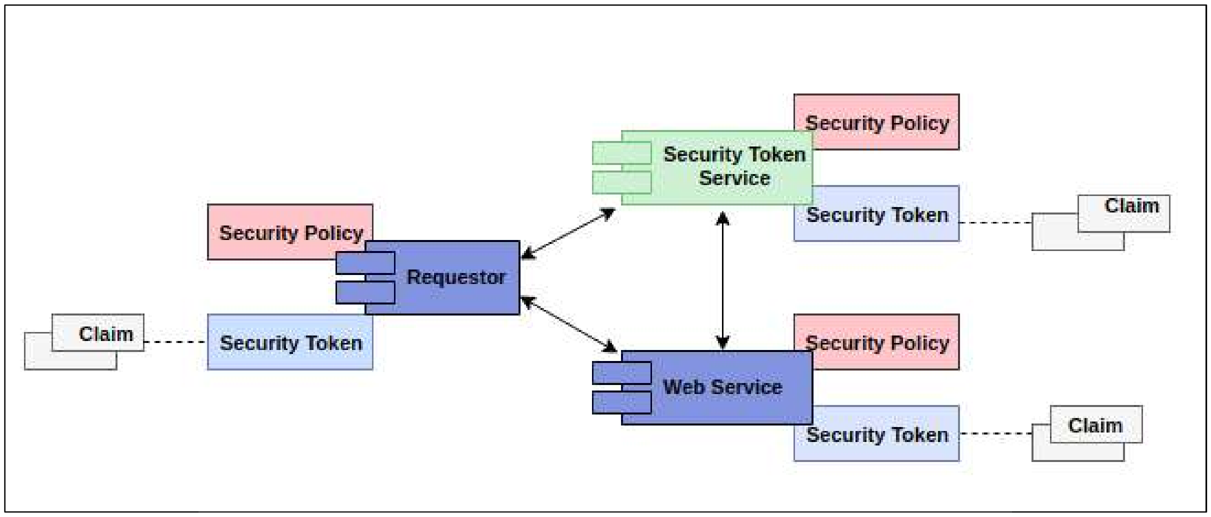


Figure 4.3 Web service security model

The goal of service-orientation requires that SOA be realized in a manner independent of technology and using open standards. The security architecture of web services should thus implement the web service security model in a manner that promotes the goal of service-orientation.

The next section discusses a concrete realization of the web services security model in the SOAP-based web services. Both SOAP and REST expose data over HTTP requests and responses, but make use of very different formats and semantics. As REST has different security considerations, REST security is discussed in the following section.

4.4 SOAP web services security

The SOAP messaging model utilizes a large combination of networks devices and applications that may be globally distributed. Web services that use SOAP messages interact by exchanging messages which may go through various intermediaries before reaching the intended destination. SOAP intermediaries are applications that can process parts of a SOAP message and forward the message as it travels from its origination point to its final destination point. Figure 4.4 shows the SOAP communication model using intermediaries. An approach called *transport-layer security* establishes a secure channel for data exchange, but is not sufficient to ensure end-to-end security in a SOAP messaging model. Intermediaries within the SOAP message path have access to messages and can, therefore, access the content of the message even when the message is not intended for them. Mechanisms are required to ensure that different parts of messages used by web services are only revealed to intended parties in the message path.

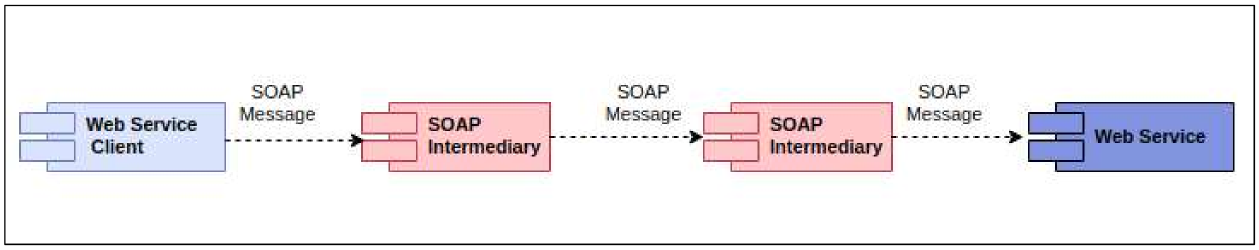


Figure 4.4 SOAP message intermediaries

Considering the nature of the SOAP messaging model and the requirement to design web services as candidates for orchestration that demands that web services are well prepared to participate in complex service compositions, the following are necessary to ensure security:

- Different parts of messages used in web services communication with information such as credit card details are protected, so that these selected parts of the message are only revealed to intended parties in the message path. This requirement is referred to as *message-level security* (Kearney, 2005, Ahmed & Bhargava, 2015, Medhi et al., 2016).
- Security rules and security enforcement mechanisms are not to be hard-coded in each web service to ensure that the web service is well prepared to participate in complex service compositions. Security requirements should instead be declared separately of the web service (Chetty & Coetzee, 2010). This approach is referred to as *policy-driven security* (Pearson & Sander, 2010, Chhetri et al., 2012).
- Components responsible for enforcing information security services on behalf of web services are required, since web services may not know the context in which they will be invoked at runtime. This approach is referred to as *security as a service* (Dawoud et al., 2010, Hussain & Abdulsalam, 2011).

The next section briefly discusses how *message-level security* and *policy-driven security* are implemented in the SOAP message model.

4.4.1 Message-level security in SOAP

Message-level security in SOAP ensures that different parts of the message have different levels of protection to ensure that only the intended party has access to sensitive information. An extension called *WS-Security* provides a means to extend SOAP message headers to address security concerns for authentication, authorization, non-repudiation, and confidentiality (Atkinson et al., 2002). WS-Security is a message-level standard that is used to secure SOAP messages using:

- XML digital signature to provide applications with authentication, data integrity and non-repudiation abilities (Rosenberg & Remy, 2004).
- XML encryption to ensure confidentiality of SOAP message (Rosenberg & Remy, 2004).
- Security tokens such as username tokens and X.509 certificates (Hawanna et al., 2016) to ensure authentication and propagation of credential between web services. WS-Security provides a means to specify and associate security tokens in SOAP messages (Rosenberg & Remy, 2004). In line with the web service security model, security tokens for use in WS-Security may be provided by other web services.

Figure 4.5 shows an example of a SOAP header message that contains an encrypted username and password using the WS-Security syntax. The example is used to enable authentication. The username and password are examples of security tokens and are contained in the *UsernameToken* element. The *UsernameToken* is an example of a WS-security token that carries a security claim. The token is encrypted to enable confidentiality.

```
<soapenv:Header>
   <wsse:Security ...>

      <wsse:UsernameToken wsu:Id="1">
      <wsse:Username>
         <xenc:EncryptedData>...</xenc:EncryptedData>
      </wsse:Username>

      <wsse:Password>
         <xenc:EncryptedData>...</xenc:EncryptedData>
      </wsse:Password>
      </wsse:UsernameToken>

    </wsse:Security>
 </soapenv:Header>
```

Figure 4.5. Encrypted SOAP header message

WS-Security allows the requester to add security information that applies to that particular message to a SOAP message header. WS-Security is enforced through SOAP message interceptors which process the SOAP request message before the web service is invoked (Taher et al., 2011, Lin et al., 2013). Typically, a web service has a security interceptor component developed according to the web service description language (WSDL) document. The interceptor acts as a security filter for incoming and outgoing SOAP messages. Interceptors exist for both the invoking web services and the web services being invoked. On the invoking web services side, the client, the client security interceptor is responsible for adding tokens on SOAP headers, signing and encrypting the SOAP message. On the side of the web services being invoked, the interceptor is responsible for verifying the message signature and checking that the message has not been tampered with in transit. When sending responses, the web service interceptor adds WS-Security headers on the response message to ensure integrity and confidentiality of the response message. Figure 4.6 show an interceptor between a web service and a client.

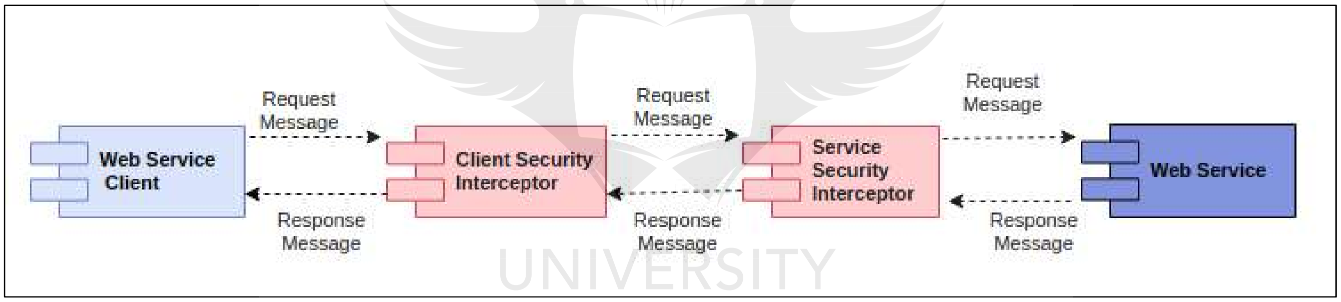


Figure 4.6. Interceptors

Securing web service compositions is complex as various services with different security requirements needs to be invoked together to get a response. Web services participating in a composition may require mechanisms to propagate authentication and authorization across composed services (Erl, 2008). Since each web service in a composition is autonomous and independent from the other, mechanisms may be required to persist the security context after authentication with the composition controller is successful.

4.4.2 Policy-driven security in SOAP web services

SOA web services should not support security as software engineers of a web service may not know the context in which the web service is to be used (Rudra & Vyas, 2015, Memeti et al., 2015). Service-orientation requires that a web service be developed in such a way that it can participate in many composition scenarios. As a result, security requirements and mechanisms should not be hard-coded in web services. Instead, security requirements of each web service need to be declared separately in a *security policy* document. A good design of a security policy is a significant requirement to successfully guarantee secure access to resources of an SOA-based application (El Hassani et al., 2015). The advantage of using a security policy in SOA is that:

- It separates security logic from business logic, leaving the former to security specialists.
- It becomes easier to ensure consistency of security enforcement across various services compositions where a web service participates.
- It enhances interoperability as security policies of web services that are a candidate for composition can be assessed at design time and decisions can be made on how to make their security implementations compatible.

A good design of a security policy requires an understanding of the following terms:

- *Subject*. A subject is an entity requesting access to a web service. A subject possesses one or more attributes. In the context of an SOA-based application, a *subject* can represent a system end user, another web service in a web service composition or a user role. A *subject* may also represent an aggregated set of users, a composition of web services and a list of roles (Brodecki et al., 2011).
- *Resource*. A resource is data, a web service or any component of the SOA system.
- *Action*. Action defines the type of access requested on a resource.
- *Obligation.* An obligation is a directive on what must be carried out before or after access is granted.
- *Target*. Target is a set of simplified condition that must be met.
- *Object.* An object is an SOA system resource for which access is managed.

An essential standard in SOA with regard to describing security policies of SOAP web services is the WS-SecurityPolicy specification (Della-Libera et al., 2002). WS-SecurityPolicy is a widely accepted security standard that allows web services to advertise their policies using XML. The standard provides means to describe a set of rules used to define security objectives to be satisfied when web services interact. This standard is part of the existing WS-Policy framework proposed for policy descriptions (Bajaj et al., 2004). WS-SecurityPolicy expresses security capability of web services using policy assertions. For example, Figure 4.7 below shows an assertion that stipulates that a request to a web service should be encrypted.

```
<wsp:Policy xmlns:wsp="..." xmlns:sp="...">
    ....
    <sp:EncryptedParts>
        <sp:Body/>
    </sp:EncryptedParts>
    .....
</wsp:Policy>
```

Figure 4.7. Example Security policy

Security policies defined using WS-policy are associated with a web service by attaching it to a web service using WS-PolicyAttachment (Box et al., 2004) or by embedding it on the WSDL file. The policy is made accessible through the UDDI registry (Bhatti et al., 2007). Association of the WS-Policy can be done at design time or at deployment time. At runtime, the security policy of a web service is enforced using a component called a policy interceptor (Lins et al., 2016, Gallino et al., 2011). When a request is made to a web service, the SOAP request message is intercepted by one or more policy interceptors defined on the invoking web services side (Jansen et al., 2015). The interceptors execute security policies that are attached to the invoking web services or client. Required security headers are added to the request message to ensure that the request message complies with the security policy of the web service or client sending the request. At the invoked web service, the request message is intercepted by the policy interceptors before it reaches the target web service. Security policies are executed, and if successful, the request message is passed to the web services. After the web services executes, a response is generated, which is intercepted by the policy interceptors, and security policies are applied before the response is sent to the requesting web service.

4.4.3 Messaging bus and security of SOAP web services

The practice of using interceptors to handle security functionality on behalf of web services assist to reduce the processing burden on the web service (Dawoud et al., 2010). Traditionally, many SOA applications have been built around a mediator called the enterprise service bus (ESB) as a component in the security model (Opincaru & Gheorghe, 2015). As part of the ESB security, mediations ensure that the appropriate message protection services are applied to all incoming web service invocation. This removes the need for each web service to independently manage and evaluate trust relationships with every possible web service invocation. Figure 4.8 below shows the ESB providing security as a service to a web services composition.

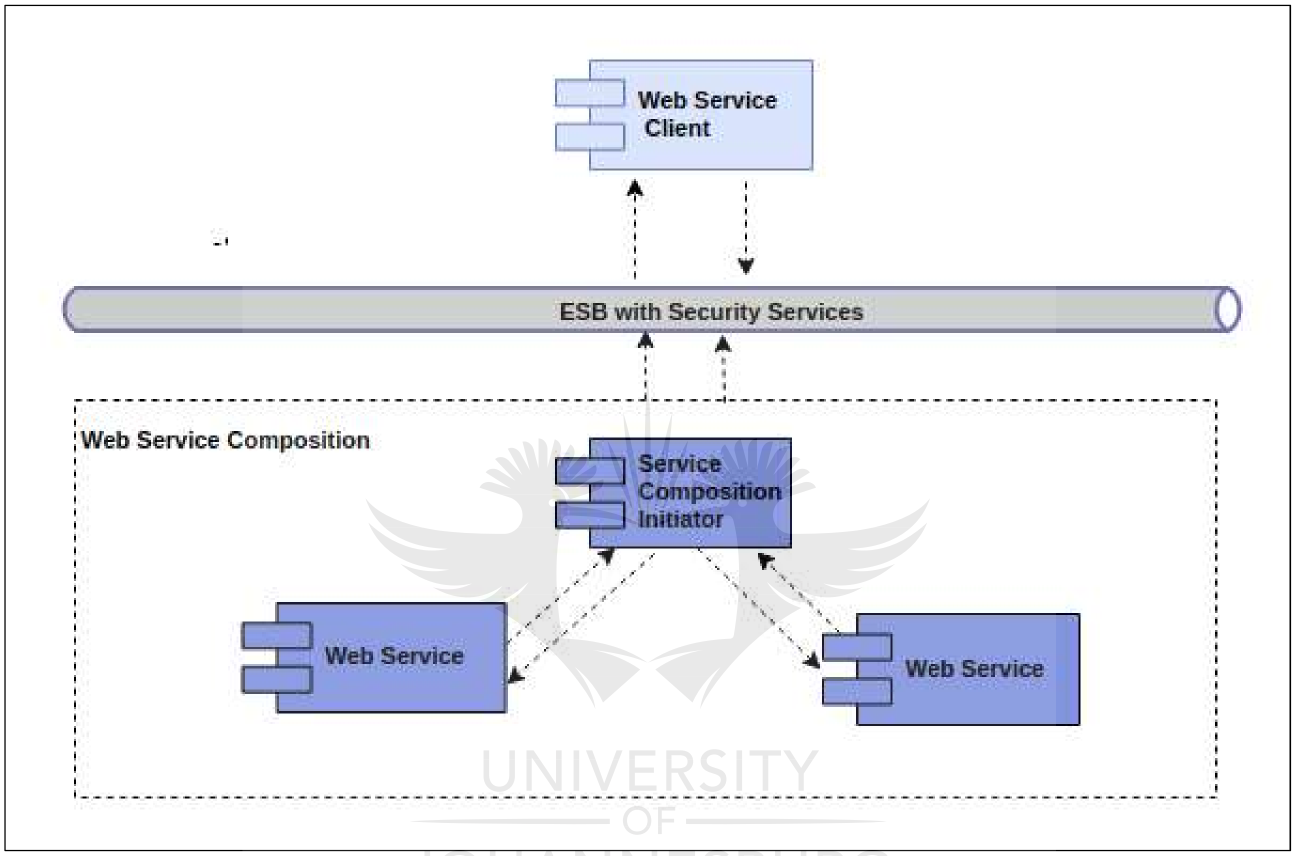


Figure 4.8. ESB and security of web services

The next section discusses a concrete realization of the web services security model in the REST messaging model.

4.5 REST web services security

REST web services lack a specific security model, unlike SOAP-based services which rely on the WS-Security standard (Kakavand et al., 2016). Most REST web services rely on transport-layer security and custom message protection mechanisms (Iacono & Nguyen, 2015). Transport-layer security offers secure point-to-point communication channels. Web services that use the REST model of communication work with Hyper Text Transfer Protocol (HTTP) Uniform Resource Locator (URL) paths and should be protected in the same manner in which websites are secured. The starting point to ensuring REST web services is implementing the below best practices (Oftedal & Stock, 2014):

- *Input validation.* Validating all inputs on the server protects REST web services from injection attacks. Validation should ensure that only well-formed input is passed to a web service.
- *No sensitive data in the URL.* Usernames, password or token should not be part of the URL because URL can easily be accessed. Sensitive values should be exchanged using the POST method.
- *Restrict method execution.* Use of methods like POST and DELETE methods should be restricted. For example, the GET method should not be allowed to delete data.

In addition to the above best practices, REST web services generally use security tokens for security. The use of a security token in the REST messaging model as a realization of the web services security model is discussed next.

4.5.1 Security tokens

REST web services can use JSON Web Tokens (JWT) (Jones, Bradley & Sakimura, 2015) as the format for security tokens for authentication and ensuring message integrity. A JWT is a JSON data structure that contains a set of claims that can be used for access control decisions (Jones, Bradley & Sakimura, 2015). JWTs are protected for integrity using a signature or a message authenticated code. The claims in a JWT are encoded as a JSON object used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed (Jones & Hildebrand, 2015). The Figure 4.9 below shows an example of the JWT payload that contains identification information, but in general, the payload can contain any information of security interest. Figure 4.9 shows an example JWT payload.

```
{
    "name": "test user",
    "email": "john@johndoe.com",
    "admin": true
}
```

Figure 4.9. Example payload of JWT

JWT is used mostly for authentication and is sent as part of the HTTP headers. Figure 4.10 below shows the use of a JWT token acquired from an authentication server. The authentication service is an example of a s*ecurity token service.* Once the user is logged in, each subsequent API request includes the JWT. The JWT allows the user to access web services, and resources that are permitted with that token.

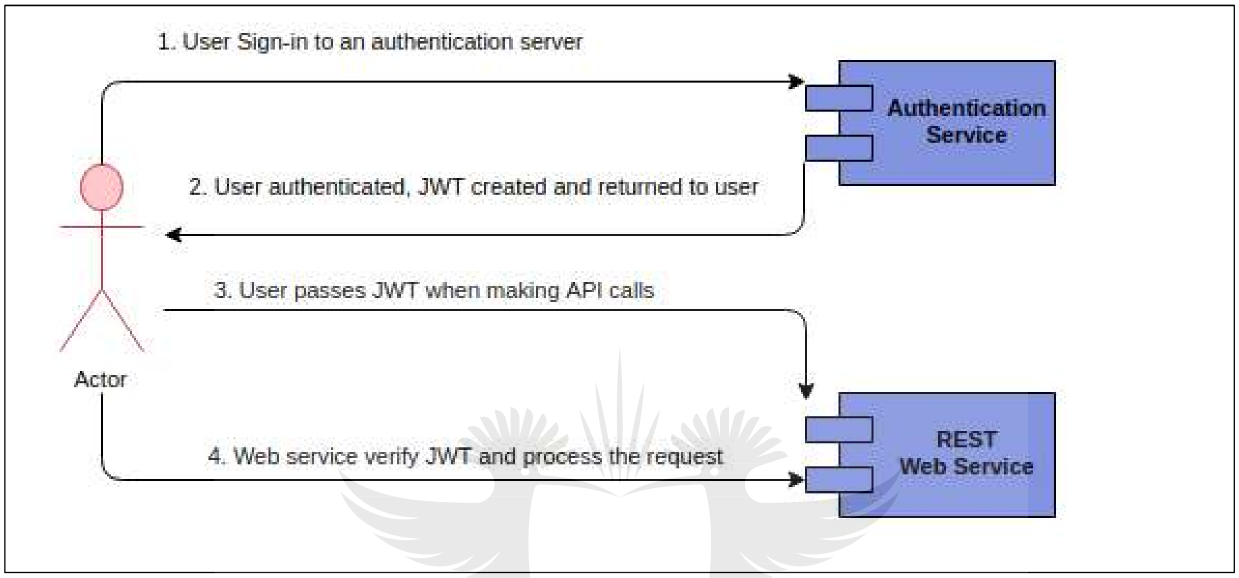

1. User Sign-in to an authentication server
Authentication Service
2. User authenticated, JWT created and returned to user
3. User passes JWT when making API calls
Actor
4. Web service verify JWT and process the request
REST Web Service

Figure 4.10. Authentication using JWT

Clients of a REST web service can also use predefined keys called application programming interface keys (API keys) to authenticate and access web services (Farrell, 2009). An API key is a piece of code assigned to a specific program, developer, or user that is used whenever that entity makes a call to an API. The key is typically a long string of generated characters which follow a set of generation rules specified by the authority that creates them. The key can be sent as part of a parameter to a URL, as part of the HTTP header or as a cookie. Figure 4.11 shows an API key sent as part of the header on the request for a trip.

```
GET /requestTrip HTTP/1.1
   X-API-Key: abcdef12345
```

Figure 4.11. API key authentication

API keys and JWT does not solve the problem of confidentiality. Therefore, a custom mechanism needs to be used to ensure data confidentiality and non-repudiation in a REST messaging model.

The adoption of the microservices architecture introduces new security challenges that cannot be addressed by the web services security model. Adoption of microservices requires a new way of thinking about security. The next section identifies the new security challenges.

4.6 Security challenges of the microservices architecture

Although microservices are a way of implementing web services, the approaches that have been used to secure traditional SOA implementations in the past are not sufficient in microservices implementations. Microservices are changing the assumptions about how SOA-based applications are created and consequently how SOA applications should be secured. The adoption of microservices architecture presents the following new security challenges.

4.6.1 Increased attack surface

When microservices are considered from a networking perspective, the instance of a microservice is a unique network endpoint with an open network port exposing an application programming interface (Thönes, 2015, Esposito, Castiglione & Choo, 2016). When a new instance of microservice is created, a new application programming interface is exposed. An attack on the microservices-based application can be made directly on each microservice (Dragoni et al., 2016). This gives the attacker an increased attack surface due to the spread of microservices instances exposed across the network. Security of microservices consequentially become a distributed security challenge.

4.6.2 Indefinable security perimeters

Many microservices are deployed in *containers* (Merkel, 2014, Stubbs et al., 2015, Amaral et al., 2015). The challenge of deploying microservices on containers is that containers can be set up quickly from anywhere within the network without any consideration for the traditional notion of demilitarised security perimeters (Combe et al., 2016, Herger et al., 2017). Containers allow port mapping functionality to masquerade standard microservices application programming interface ports to dynamically allocated ones (Marmol et al., 2015). The use of dynamic addressing and scaling of microservices makes it a challenge to statically configure internet protocol addresses or steer network traffic to traditional perimeter security appliances.

4.6.3 Security monitoring is complex

Containers present a security monitoring challenge. Containers on a host machine can use *network address translation* (NAT) which makes them invisible to the outside world (Anderson 2015). *Network address translation* is the process where a network device, usually a firewall, assigns a public address to a computer inside a private network. Network traffic from containers using NAT is challenging to identify. When containers use NAT, a definition of security policies becomes complicated because it becomes difficult to know which microservices is running in each container. Containers may also bundle applications with a lot of software libraries and files that software engineers may not be aware of (Pittenger, 2016). This may increase the security risk due to vulnerabilities that may be hidden inside the software libraries (Moradi et al., 2017).

4.6.4 Authentication is centralised

Microservices deployed in containers interact remotely, mostly over HTTP. The challenge of this approach is how users of microservices are authenticated and how user credentials are passed between microservices in a symmetric manner (Esposito et al., 2016). Another challenge when ensuring inter-microservices communication between a large number of microservices is that when microservices use transport layer security, certificate revocation becomes a harder problem (Yarygina, 2017). The microservices that initiate the handshake may get a list of revoked certificates from the corresponding certificate authority which can grow bigger.

4.6.5 Threat modeling and risk assessment is localised

The emphasis on team autonomy makes it challenging to ensure that threat modeling, and risk assessment is done before new versions of microservices are released (Ur, Ashfaque & Williams, 2016). Continuous delivery can mean that new vulnerabilities are delivered with every new microservices deployment (Wilde et al., 2016).

The five challenges above provide an answer to research question RQ2 formulated in Chapter 1. Considering the challenges listed above, the adoption of microservices, therefore, require new ways of ensuring security.

4.7 Conclusion

Building traditional SOA-based and microservices-based applications that are secure is a complex exercise. Security plays an essential part in the decision to adopt SOA, also when adopting microservices architecture as a concrete implementation of SOA. Since traditional mechanisms of securing SOA-based applications are not sufficient within a microservices architecture, new mechanisms are required to ensure secure microservices-based SOA implementations. Still, the problem of how to provide web service-based systems that are secure by default is an unsolved research challenge.

The security challenges of microservices discussed above demand that security should be an integral part of the architecture. Securing microservices requires that security requirements be incorporated from the beginning and security be made part of the microservices build, test, and delivery chain. The challenge of an increased attack surface and complex networking resulting from the communication model of microservices requires a new security strategy to mitigate this new threat. The rapid approach to designing and deploying microservices requires a new security design and testing approach to ensure that no new vulnerabilities are introduced with each deployment. Monitoring mechanisms are required to monitor the communication paths and containers. When changes that may affect the security of the microservices-based application are detected at runtime, changes to microservices security configuration is required to stay up to date with a change that may occur on the underlying deployment environment.

To this end, the next chapter discusses a preliminary risk analysis of the microservices architecture to provide an understanding of security threats, the potential attacker's profile, the most likely attack vectors and the assets most likely to be targeted by an attacker. This knowledge is useful to ensure that microservices-based applications are designed to avoid vulnerabilities and to withstand any attack.

PART II

Chapter 5

Software Development Activities for Secure Microservices

5.0 Introduction

Organizations face countless challenges in the business world as they need to be able to counter disruptive competitors, adapt to new trends, be highly flexible, and provide engaging customer experiences. The adoption of Agile methodology and continuous delivery practices, by employing microservices architectures is therefore essential (Bossert 2016). However, the decomposition of an application into a set of distributed and collaborating microservices, using microservices architecture principles, increases an application's attack surface.

Furthermore, the continuous delivery practices increase the rate of releasing new software changes, leaving little time to identify and understand potential or actual adversary loopholes that can be introduced into a microservices composition through each new deployment. On the contrary, malicious attackers have unlimited time and resources to devise ways to attack microservices compositions. It, therefore, becomes vital to carry out a preliminary security risk analysis at design-time of the entire microservices composition.

A preliminary risk analysis provides an understanding of security threats from a hypothetical attacker's point of view. Identified security threats equip software engineers of microservices compositions with knowledge of assets most likely to be targeted, the most likely attack vectors, and the potential attacker's profile. The knowledge is useful to ensure that microservices compositions are designed to avoid vulnerabilities and to withstand any attack, and in the event of an attack to ensure that adverse consequences of an attack are minimized.

In this regard, this chapter aims to identify security threats that could arise as a result of flaws in the design of microservices compositions and harm that may arise from misuse of a microservices composition by malicious users. The preliminary risk analysis leads to a list of security requirements to be met by this research to be able to develop secure microservices compositions. The contribution of this review is a list of development activities for secure microservices.

This chapter is organized as follows; Section 5.1 introduce secure development frameworks. Section 5.2 introduces threat modeling to identify, enumerate and prioritize security threats. Section 5.3 discusses threat modeling of a microservices composition. Section 5.4 uses the outcomes of threat modeling to elicit security requirements of a microservices composition. Section 5.5 Suggests protection mechanisms and techniques that can be used to satisfy the security requirements obtained from Section 5.4. A list of software development activities to ensure the implementation of secure microservices compositions is then derived. A conclusion then follows in section 5.6.

5.1 Secure software development frameworks

To date, the construction of secure software is directed by a number of guidelines, best practices and undocumented expert knowledge such as blogs and discussions. There exist a number of best practices for areas such as threat modeling, risk management, or secure coding (De Win et al., 2009) and various new approaches to support the development of secure software in agile environments. To be able to support the development of secure applications, it is very important that these aspects are combined into an integrated and more comprehensive construction method. Traditionally, sequential software development approaches integrate security engineering activities commonly defined in a sequence, where security verification and validation gates are created for each of the development stages of analysis, design, coding, and testing.

Frameworks that are used to develop secure software depend on a risk assessment being conducted to identify weaknesses in the software (Shostack, 2008). The identified weaknesses are used to elicit security requirements of the system and also to guide the creation of secure designs. Furthermore, software engineers identify protection measures from the identified risks and incorporate these measures in the system during the development. Testing of the system is used to ensure that the risk is mitigate using identified protection measures.

The next section discusses threat modeling to assess the security risk of microservices for this research.

5.2 Security threat modeling

Various security weakness or vulnerabilities possibly exist on microservices and their runtime infrastructure that an attacker can exploit to breach the security of microservices-based applications. To be able to understand the security of microservices compositions, the composition needs to be analysed from the perspective of a potential attacker. The process of identifying and documenting security threats is called *threat modeling* (Shostack 2008). Threat modeling provides a good foundation to identify security requirements (Myagmar, Lee & Yurcik, 2005). This type of view can be gained by first identifying, enumerating and prioritizing security threats of a microservices composition.

Threat modeling identifies entry points into a system and the associated threats that each entry point exposes. They are three main approaches to threat modeling namely (Shostack 2008):

- *Architecture-centric threat modeling* - The architecture-centric threat modeling approach also called system-centric, or design-centric approach focuses on the design of a system. The approach attempts to step through the components of the system identifying the potential types of attacks against each component (Martins et al., 2015).
- *Asset-centric threat modeling* - The asset-centric threat modeling approach starts by identifying and quantifying the value of vital assets in a system and the motivation of the attacker (Rauter, Kajtazovic & Kreiner, 2016).
- *Attack-centric threat modeling* - Attack-centric threat modeling starts by identifying the goals of an attacker and the possible techniques that the attacker might use to achieve malicious goals (Tuma et al., 2017).

The next section uses threat modeling to identify security threats in microservices compositions where microservices compositions are realized by a set of collaborating microservices.

5.3 Microservices composition threat modeling

Architecture-centric threat modeling is used next to identify security threats in a microservices composition. The architecture-centric threat modeling approach is more suitable in this context as it provides a means to step through the components of a microservices composition to identify potential types of attacks against each component. The *PickMeUp* microservices composition discussed in previous chapters is used as an example of a microservices composition. The architecture-centric threat modeling steps from the Microsoft threat modeling process is followed, as shown in Figure 5.1 as per microservices compositions (Priya & Arya, 2016).

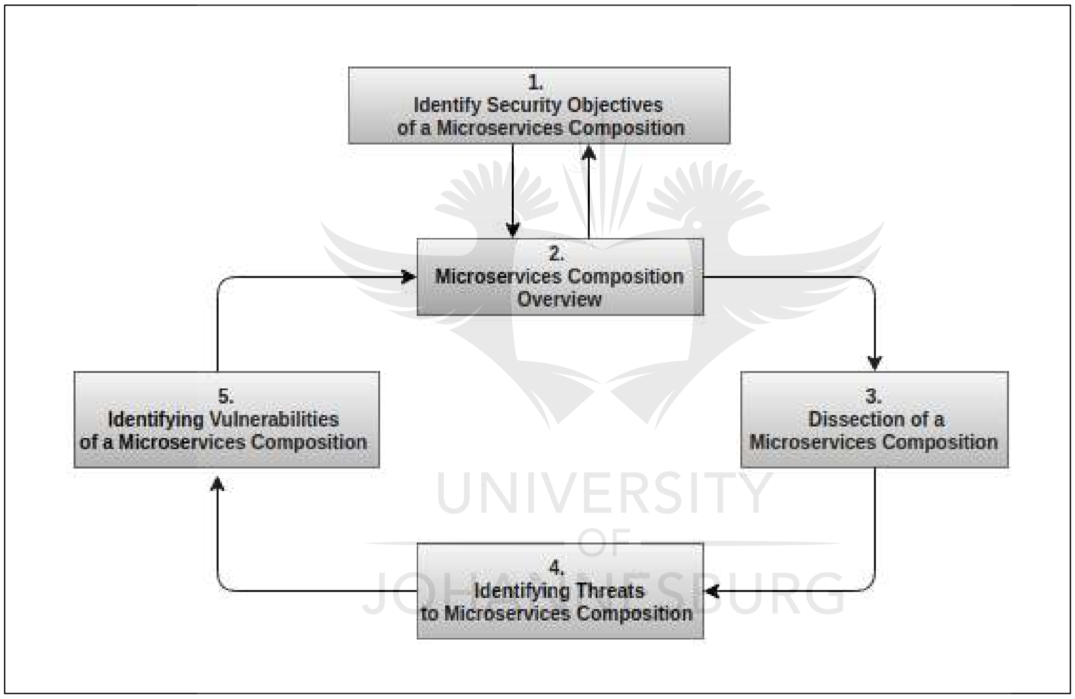

1. Identify Security Objectives of a Microservices Composition
2. Microservices Composition Overview
3. Dissection of a Microservices Composition
4. Identifying Threats to Microservices Composition
5. Identifying Vulnerabilities of a Microservices Composition

Figure 5.1. Microservices threat modeling steps

Next, the first step of threat modeling, shown in Figure 5.1, is performed where the security objectives of a microservices composition are identified.

5.3.1 Microservices composition security objectives

A microservices composition is a set of collaborating microservices. Each microservice is an open system similar to any web service, and therefore needs to address security services as prescribed by ISO 7498-2 namely:

- Authentication
- Access control
- Data confidentiality
- Data integrity
- Non-repudiation

In addition to identifying security objectives, an essential step in the architecture-centric threat modeling approach is to determine the components of the application as a foundation towards the elicitation of potential types of attacks. In this regard, the next section provides an overview of the architecture of a microservices composition and list the essential components**.**

5.3.2 Microservices composition overview

The second step of threat modeling as shown in Figure 5.1 uses the PickMeUp application shown in Figure 5.2 to illustrate essential components that any microservices architecture application generally would consist of as follows:

- *The API gateway* - a lightweight entry point into an application.
- *A set of microservices* - components that automate business functionality.
- *Service registry* - a database of instances and locations of all active microservices in a microservices composition.
- *Message broker* - used by microservices in composition to publish and receive messages.
- *Containers or virtual machine* - provide the runtime environment to microservices.

To find threats in microservices, sources of threats and specific components of the application that may be affected should be known. The next section performs the third step of the architecture-centric threat modeling to identify parts of PickMeUp that are potential sources of threats.

5.3.3. Decomposition of a microservices composition

The various architectural components of the PickMeUp application are shown in Figure 5.2 below. Steps 1 to 13 shows the flow of information from when a request for a trip is received from a passenger using a mobile device until the final response is sent. For the sake of brevity, information flow for payment is not shown. The deployment strategy adopted for the application is to deploy components of the application on separate Docker containers. The containers run on a single host.

When Figure 5.2 is viewed from the attacker's perspective, the following architectural components provide potential entry points to maliciously access and compromise the PickMeUp microservices composition:

- *The API gateway and the microservices API* - the attacker may use the gateway and microservices API to perform various types of injection attacks.
- *The service registry* - the attacker may control the service registry to compromise the microservices composition or to shut the microservices composition down by ensuring that collaborating microservices cannot locate one another.
- *Message broker* - the attacker may gain access to messages exchanged by microservices or to bring the message broker down so that the composition ceases to function.
- *Container or virtual machine* - the attacker may gain control of the runtime environment where the application is running and control or shut down the microservices composition.

The four entry points listed above in general form the attack surface of any microservices composition. Considering the technical design and implementation choices made during the development of the API gateway, service registry, message broker, containers or virtual machines the following security threats are now derived by the researcher from the four entry points listed above:

- Insecure application programming interfaces
- Unauthorized access
- Insecure microservice discovery
- Insecure runtime infrastructure
- Insecure message broker

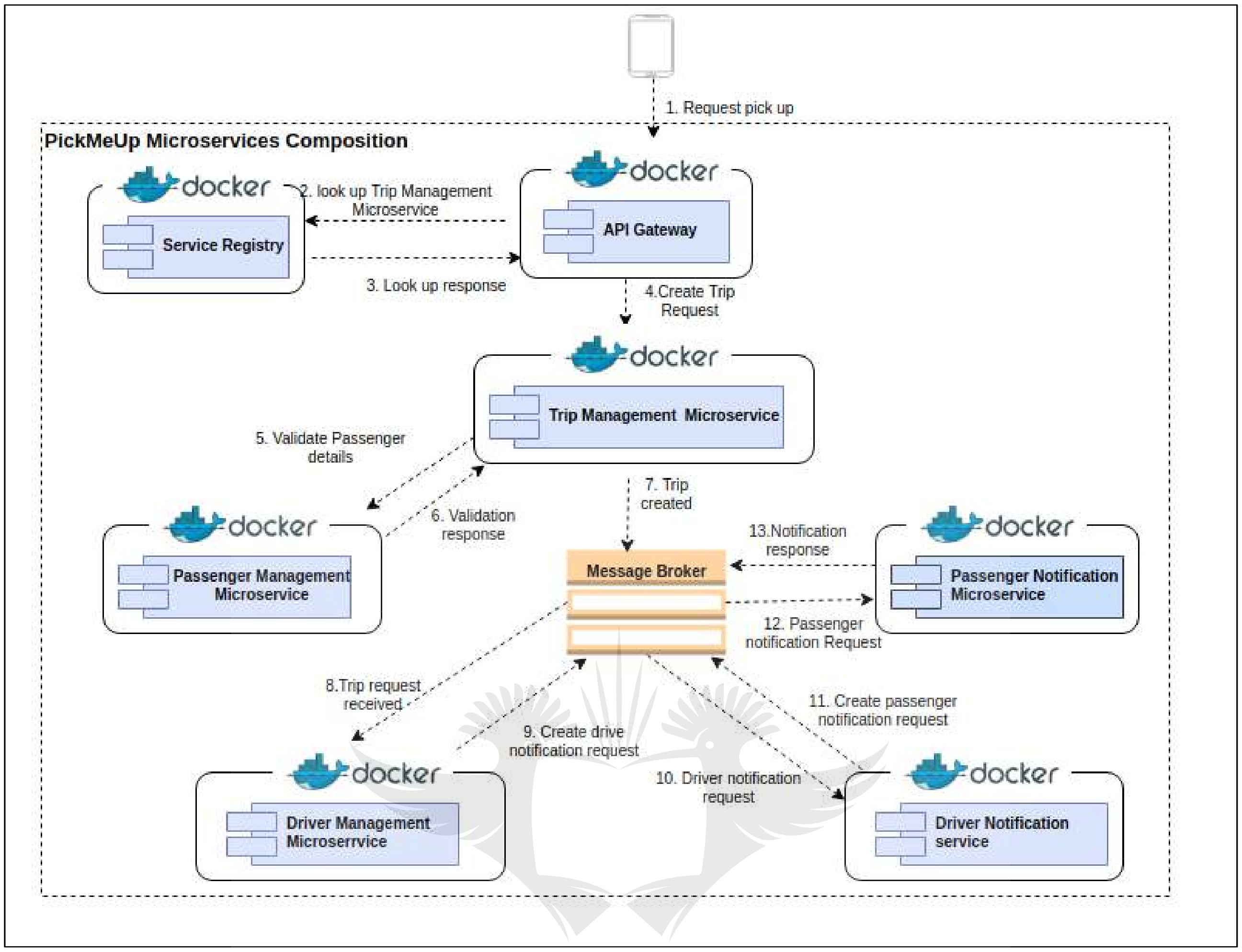

1. Request pick up
PickMeUp Microservices Composition
docker
Service Registry
2. look up Trip Management Microservice
API Gateway
3. Look up response
4.Create Trip Request
Trip Management Microservice
5. Validate Passenger details
6. Validation response
7. Trip created
Passenger Management Microservice
Message Broker
13.Notification response
Passenger Notification Microservice
12. Passenger notification Request
8.Trip request received
9. Create drive notification request
11. Create passenger notification request
10. Driver notification request
Driver Management Microserrvice
Driver Notification service

Figure 5.2. PickMeUp Microservices composition

The next section discusses each security threat in more detail. Attention is given to the methods that an attacker can exploit to compromise the microservices composition. First, a security threat classification model is presented as a foundation towards understanding each threat.

5.3.4 Security threats classification

Conceptually, threat modeling is performed by applying a methodology (Shostack 2014). In this regard, each security threat identified above is reviewed using a threat categorization model developed by Microsoft called STRIDE (Shostack 2014, Scandariato, Wuyts & Joosen, 2015). Even though there are many threat modeling tools available, the researcher chose STRIDE as it offers a very systematic approach to analyse threats against each of the microservices architectural components. STRIDE provides a clear understanding of how an identified vulnerability can impact the whole system and supports a comprehensive review of security services such as authentication, authorization, confidentiality, integrity, nonrepudiation, and availability. STRIDE is an acronym that stands for:

- *Spoofing* - an attempt by an attacker to gain access to an application using false identity (Shostack, 2014).
- *Tampering* - the unauthorized modification of information or data (Shostack, 2014).
- *Repudiation* - the ability of an attacker to deny an action that has been performed (Scandariato, Wuyts & Joosen, 2015).
- *Information disclosure* - when private data is revealed to an unintended user (Scandariato, Wuyts & Joosen, 2015).
- *Denial of service* - the process of making an application unavailable to legitimate users (Shostack, 2014).
- *Elevation of privilege* - occurs when a user with limited or no privileges assumes the identity of a privileged user to access an application (Shostack, 2014).

STRIDE allows characterizing of identified threats and provides a method to reason about each security threat, and to determine potential exploits that can be used by an attacker. The five security threats identified by the researcher in the previous step are now discussed in detail using the STRIDE model. This section considers the attack methods associated with each security threat and the vulnerabilities that make the attack possible.

a) Insecure application programming interfaces

A weak set of APIs exposes microservices to a variety of security attacks that may result in tampering with data, information disclosure, denial of service and elevation of privileges (Cloud Security Alliance, 2017). Table 5.1 below list the attack methods and weaknesses on the composition that can make the attack possible.

Table 5.1. STRIDE analysis of insecure application programming interfaces

Security threats (STRIDE)	**Attack methods**	**Exploitable weaknesses or known Vulnerabilities**
Tampering with data	• Intercept and modify messages sent to a microservices API when communication channels used is not secured. • Exploit vulnerability in mechanisms used for transport-layer security. • Perform all forms of injection attacks on the API.	• Insecure communication channel • Lack of mechanisms to protect against injections of all forms on the APIs • Weak access control schemes on the microservices API • Vulnerability CVE-2014-3566 (Sheffer, Holz & Saint-Andre, 2015) that allows an attacker to obtain clear text when Secure Socket Layer (SSL) v3.0 is used.
Information disclosure	• Perform all forms of injection attacks on the microservices API. • Exploit weak access control schemes used to protects APIs	• Lack of mechanisms to protect against injections of all forms on the APIs • Weak access control schemes on the API. For example, the United States of America Internal Revenue Service (IRS) exposed over three hundred thousand (300 000) customer records using a vulnerable web API (Borazjani, 2017). • Vulnerability CVE-2017-9805 (NIST, 2017) in the REST plugin of a web application framework called Struts. The vulnerability resulted in the Equifax data breach were an attacker gained access to consumer credit reports of about one hundred and forty-three (143) million United States citizens (Gressin, 2017, O'Brien, 2017).
Denial of Service	• Craft a request to API gateway that fans out into multiple computationally expensive requests to microservices behind the gateway so that microservices slow down and impact all legitimate users (Behrens & Heffner, 2017).	• Failure to prioritize authenticated traffic over unauthenticated traffic • Lack of reasonable microservices requests time-outs • Lack of fallback options when microservices does not respond on time • Lack of fault isolation mechanisms.
Elevation of privileges.	• Exploit a parser of messages used on the API that allow deserialization of hostile or tampered objects by changing the serialized object to gain administrative privileges.	• Insecure message deserialization

In addition to the threat of insecure application programming interfaces, microservices are exposed to the threat of unauthorized access, discussed next.

b) Unauthorized access

When there is no proper scalable identity access management system, a microservices composition is vulnerable to unauthorized access (Cloud Security Alliance, 2017). Unauthorized access can lead to tampering with data and information disclosure. Table 5.2 below list the attack methods and weaknesses on the microservices composition that can make the attacks possible.

Table 5.2. STRIDE analysis of the threat of insecure APIs

Security threats (STRIDE)	**Attack methods**	**Exploitable weaknesses or known vulnerabilities**
Tampering with data	• Use harvested login credentials to gain access and tamper with data. • Gain administrative access to a microservices runtime environment that use single-factor authentication mechanism and destroy data.	• Insecure management consoles. For example, an online hosting and code publishing provider called Code Space went out of business when an attacker gained access to the company's Amazon Web Service's (AWS) control panel account and destroyed customer's data (Cloud Security Alliance, 2017). The Amazon Web Service environment is a popular platform for running microservices. • Lack of scalable identity access management • Lack of multi-factor authentication
Information disclosure	• Use methods such as phishing and fraud to gain access to credentials that are often re-used.	• Cryptographic keys and passwords embedded in software source code that is in public facing software repositories. For example, an attacker accessed a software repository used by Uber software engineers and used the login credentials to access Uber customer data (Newcomer, 2017, Giles, 2017). • Lack of scalable identity access management • Lack of multi-factor authentication

The use of services registries in a microservice may pose a threat of insecure microservice discovery, discussed next using the STRIDE categorization model.

c) Insecure microservices discovery

When microservices use discovery mechanisms that are not secure spoofing, information disclosure and denial of service may occur. Table 5.3 below list the attack methods and weaknesses on the microservices composition that can make the attack possible.

Table 5.3. STRIDE analysis of the threat of insecure microservices discovery

Security threats (STRIDE)	**Attack methods**	**Exploitable weaknesses or known Vulnerabilities**
Spoofing	• Intercept microservices registration requests sent to a service registry to gain access to private information about a microservice.	• Insecure communication channels • Insecure certificate distribution
Information disclosure	• Gain access to microservices identity during microservices lookup queries.	• Insecure communication channels between a microservice and the service registry.
Denial of service attack	• Flooding registration messages to the service registry to force the service registry to consume and exhaust its resources and ultimately become slow or stop functioning. • De-register a microservice from the registry by sending a bogus de-registration message to the service registry.	• Messages for registration or de-registration sent without integrity protections • Lack of message verification

In addition to insecure microservice discovery, the runtime infrastructure where microservices are deployed may pose a security threat, discussed next using the STRIDE categorization model.

d) Insecure runtime infrastructure

Containers and virtual machine, where microservices are deployed, may be compromised by the presence of errors or malware on the infrastructure that an attacker can exploit to infiltrate microservices compositions. Vulnerabilities in the microservices runtime infrastructure may result in spoofing, information disclosure, denial of service, and elevation of privileges (Cloud Security Alliance, 2017). Table 5.4 below list the attack methods and weaknesses on the composition that can make the attack possible.

Table 5.4. STRIDE analysis of insecure runtime infrastructure

Security threats (STRIDE)	**Attack methods**	**Exploitable weaknesses or known vulnerabilities**
Spoofing	• Sniff secrets like cryptographic keys, certificates, and passwords. • Gains control of a container and receive, redirect and manipulate information being delivered to containers, an attack called Address Resolution Protocol (ARP) spoofing (Scott et al., 2017). • Inject a malicious payload into network connections.	• An insecure configuration of containers and virtual machines and using default container settings that allow open communication such as in Docker containers (Scott et al., 2017). • Improper user access rights • Insecure communication channels
Information disclosure	• Inject malicious code into the containers to gain access to sensitive information.	• Vulnerability CVE-2014-6271 (Shetty, Choo, & Kaufman, 2017) also known as shellshock that allows one to inject malicious code into the command line interface connects users to Unix-based systems. • Host operating system vulnerabilities • Runtime software vulnerabilities
Denial of service attack	• Create malicious image payload which may require an inordinate amount of time, disk space and memory to decompress in such a way that decompressing images result in denial of service.	• Lack of content verification on containers. • Vulnerability CVE-2017-14992 (Redhat, 2017) which allows a remote attacker to cause a denial of service by using a crafted docker image payload on Docker-CE and all earlier Docker versions.
Elevation of privileges.	• Escape from the confines a compromised virtual machine and obtain elevated access to the host machine, the host's local network and adjacent systems. Elevate access privileges and cause damage to microservices running on the host.	• Vulnerability CVE-2015-3456 (Brook & Brooks, 2015) also called Virtualized Environment Neglected Operations Manipulation (VENOM) allows an attacker to escape from the confines a compromised virtual machine and potentially obtain elevated access to the host machine, host's local network, and adjacent systems.

An attacker can also leverage the decoupled nature of publish-subscribe message brokers used in microservices compositions, discussed next.

e) Insecure Message Broker

When the message broker is not correctly secured spoofing, tampering with data, information disclosure and denial of service may occur. Table 5.5 below list the attack methods and weaknesses on the composition that can make the attacks possible.

Table 5.5. STRIDE analysis of the threat of insecure message broker

Security threats (STRIDE)	**Attack methods**	**Exploitable weaknesses or known vulnerabilities**
Spoofing	• Intercept and modify messages sent to the API when communication channels used is not secured. • Exploit vulnerability in mechanisms used for transport-layer security.	• CVE-2014-3566 (Sheffer, Holz & Saint-Andre, 2015) that allows an attacker to obtain clear text data against any application that uses Secure Socket Layer (SSL) v3.0.
Tampering With data	• Modify stored messages or messages in transit in the publish-subscribe model of communication. • Tamper with data by maliciously changing messages exchanged by microservices. • Exploit message parsers that perform insecure message deserialization by sending crafted messages to perform remote code execution or to create recursive objects graphs.	• Unsafe deserialization. Message broker relies on message parsers to function. A parser in a message broker calls Spring AMQP (Gutierrez, 2017) was found with vulnerability CVE-2017-8045 (Pivotal, 2017) that allows unsafe deserialization of message. Unsafe deserialization is listed among the top ten most critical web application security risks by OWASP in 2017 (OWASP, 2017).
Information disclosure	• Read sensitive messages in transit between publisher-broker-subscriber or at rest when the channel of communication used by microservices is not secured	• Vulnerability CVE-2017-9805 (NIST, 2017) found in the REST plugin of a web application framework called Struts. The vulnerability resulted in The Equifax data breach were an attacker gained access to consumer credit reports of about one hundred and forty-three (143) million United States citizens (Gressin, 2017, O'Brien, 2017).
Denial of Service	• Set up a disguised microservices that publish many large messages to the broker to exhaust the resources of the message broker. • Exploit mishandled exceptions that arise in a message being queue again after it has been dequeued. This may potentially exhaust the resources of the message buffer that stores the message in a manner that no new messages are accepted.	• Open message publishing • Open message subscription

Table 5.6 gives a high-level summary of a list of flaws associated with each security threat identified above. The software weaknesses are extracted from Tables 5.1 – Table 5.5.

Table 5.6. *Summary of microservices vulnerabilities*

Security threats	**Security vulnerabilities**
Insecure application programming interfaces	• Lack of mechanisms to protect against injections of all forms of APIs • Weak access control schemes • Insecure message deserialization • Insecure communication channel
Unauthorized access	• Lack of scalable identity access management • Lack of multi-factor authentication
Insecure microservice discovery	• Insecure communication channels • Insecure registration and certificate distribution • Registration messages sent without integrity protections • Lack of registration message verification
Insecure runtime infrastructure	• Improper user access rights • Host operating system vulnerabilities • Runtime software vulnerabilities • Insecure container or virtual machine configuration • Poisoned container images
Insecure message broker	• Open message publishing • Open message subscription • Insecure message deserialization

In the next section, the five security threats and security flaws in Table 5.6 are used in a systematic manner to elicit a set of general security requirements common to most implementations of microservice compositions. First, a definition of security requirements is provided.

5.4 Microservices compositions security requirements

Security requirements for a microservices composition describe more concretely the conditions and capabilities that must be met or be possessed by the microservice composition to assure the security of assets. The Open Security Alliance (Open Security Alliance, 2003) distinguishes four different types of security requirements:

- *Secure functional requirements* - describe the security services that should be integrated into each functional requirements of the system (Jain & Ingle, 2011). They specify what should not happen on the system during execution. The requirements are elicited from identifying abuses to the system referred to as misuse cases (Open Security Alliance, 2003).
- *Functional security requirements*- describe the functional behaviors that enforce the security of the system under inspection (Jain & Ingle, 2011).

- *Non-functional security requirements*- security related architectural requirements, such as robustness and scalability. They are typically derived from architectural principles and best practices or standards (Jain & Ingle, 2011). Non-functional security requirements are considered out of scope in this chapter.
- *Secure development requirements* - describe the activities required during system development which assure that the outcome is not subject to security flaws (Open Security Alliance, 2003).

Security requirements of interest for this research are s*ecure functional requirements, functional security requirements and secure development requirements.* As shown in Figure 5.3 this chapter elicits the security requirements of a microservices composition from the following:

- Security needs of the user of a microservices composition
- Microservices composition security threats
- Best practice and standards
- Regulations and laws

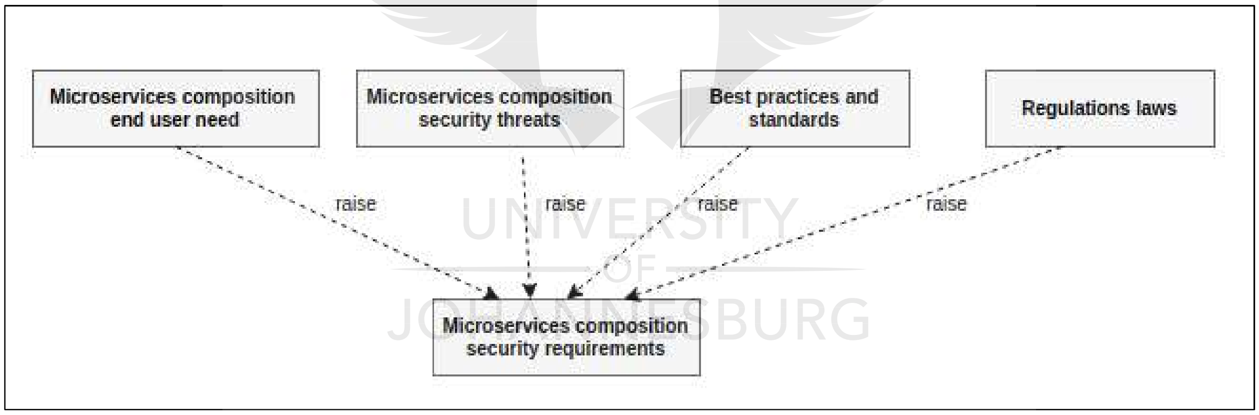

Microservices composition end user need
Microservices composition security threats
Best practices and standards
Regulations laws
raise
raise
raise
raise
Microservices composition security requirements

Figure 5.3. Elicitation of security requirements of a microservices composition

Using the five security threats and security flaws listed in Table 5.6 above, Table 5.7 below derives and documents security requirements for a secure microservices composition. Satisfying the security requirements should lead to more secure microservices compositions. The approach adopted in this section to elicit security requirements is to consider such requirements as constraints that limit the manner in which the microservices composition is developed, and how each component of the composition (the microservices, the message broker, the service registry, the API gateway, and the runtime infrastructure) should function to ensure overall security. The security requirements are expressed as positive statements to help verify their satisfaction (Haley et al., 2008). Also, Table 5.7 suggests a list of protection measures that can be used to satisfy the security requirements.

Table 5.7. Microservices composition security requirements and protection measures

Security threats	**Security requirements**	**Suggested protection measures**
Insecure application programming interfaces	• Only authenticated users should access the API • Keys, tokens, and password should be rotated periodically • The API should validate all requests. • The communication channel between microservices should be secure	• Use keys or security tokens or passwords to protect API • Perform input validations on the microservices API • The API should white-list permitted HTTP methods • Ensure secure management of keys, password, and tokens • Use transport-layer security • Monitor the microservices API at all times
Unauthorized access	• Access to microservices should be denied by default • The microservices composition should use multi-factor authentication at all entry points • Any credentials used in the microservices composition should be rotated periodically	• Use keys, security tokens, and password to protect API • Use transport-layer security • Automate management of keys, password, and tokens
Insecure microservice discovery	• The service registry should authenticate all requests for registration • Communication between microservices and service registry should use a secure channel • Messages for registration and de-registration should be protected for integrity	• Ensure the host on which the service registry run is securely configured • The service registry should use certificates and certificates should be distributed securely • Use transport- layer security • Monitor the service registry at all times
Insecure runtime infrastructure	• Containers and virtual machines should only use verified operating systems platforms or container-specific operating systems • The outbound network traffic sent by container should be monitored and controlled • The configuration of containers and virtual machines should comply with the configuration standards	• Create secure configurations of infrastructure • Validate the configurations of infrastructure • Scan container images before deployment • Group containers by relative sensitivity and only run containers of a single sensitivity level on a single host • Monitor infrastructure at all times

Security threats	**Security requirements**	**Suggested protection measures**
Insecure message broker	• The message broker should authenticate all requests • A secure channel should be used for communications • The client should protect the message it sends for integrity. • A redundancy mechanism should be configured to guarantee the delivery of the messages	• User transport-layer security • Use authentication plugins or write a custom filter to authenticate a message • Set up read and write permissions on the message broker • Monitor the service registry at all times

The list of security requirements and the protection measures in Table 5.7 above points to the need to integrate security in different phases of the software development lifecycle such as requirements gathering, design, implementation, and testing (ben Othmane et al., 2014). To achieve this, various security-focused activities are required during the different development phases of microservices to assure that the microservice composition is not subject to security vulnerabilities. The Open Security Alliance refer to such security-focused activities as *secure development activities* (Open Security Alliance, 2003). In this regard, the next section uses Table 5.7 to identify the essential security-focused activities that should be incorporated into the development process of microservices and microservices compositions.

5.5 Software development activities for secure microservices compositions

Using the security requirements and suggested protection measures in Table 5.7, this section derives a list of software development activities that can be used to ensure that microservices are adequately protected. Table 5.8 below gives a list of security-focused activities to address each of the five security threats.

Table 5.8. Secure microservices composition development activities

Security threats	**Security-focused activities**
Insecure application programming interfaces	• Document security requirements for the microservices API at design time • Adopt secure programming best practices for input validations on the API and white-listing permitted HTTP methods • Validate the implementation of the API during the continuous delivery phase for adherence to secure coding standards and security requirements • Monitor the API continuously at runtime

Security threats	Security-focused activities
Unauthorized access	• Document microservice API access requirements • Validate the API for adherence to access requirements during the continuous delivery phase
Insecure microservice discovery	• Document of security requirements for microservice discovery at design time • Validate security requirements for service discovery during continuous delivery phase • Monitor the service registry at all times
Insecure runtime infrastructure	• Documentation of security requirements for runtime infrastructure • Create a secure configuration of runtime infrastructure • Test the infrastructure for adherence to security requirements • Monitor the infrastructure at all times
Insecure message broker	• Document of security requirements for message broker • Validate the message broker for message broker adherence to security requirements • Monitor the message broker at all times

The security-focused activities in Table 5.8 above can generally be summaries into the following six secure development requirements, prefixed with *SDA*:

1. *Document security requirements of microservices compositions (SDA-1)* – security of microservices compositions requires a flexible way to manage security requirements and protection measures. Documentation is vital to provide a security strategy to secure various assets of the microservices (Terala & Cole, 2015).
2. *Adopt secure programming best practices (SDA-2)* - some of the protection measures suggested in Table 5.7 to protect the API are documented guidelines to mitigate known assaults on many web applications. A vital activity is to ensure that engineers adopt these guidelines to ensure that microservices are designed and implemented to avoid vulnerabilities. Examples of such documented instructions include validating inputs, ensuring that the application executes with the least set of privileges required for the job and sanitizing any data sent to other application to avoid injection attacks.
3. *Validate security requirements and secure programming best practices (SDA-3)* – validating the implementation of a microservices composition against a set of security requirements and security coding standards at various stages of the microservices' build, test and deployment is vital to ensure end-to-end security (Ciuffoletti, 2015, Callanan & Spillane, 2016).

4. *Secure configuration of runtime infrastructure (SDA-4)* - a vital security activity is to ensure that containers and the virtual machine are securely configured in an automated manner. Such configuration eliminates human errors that may result in misconfiguration.
5. *Continuously monitor the behavior of components of the microservices composition (SDA-5)* - ensuring continuous security of microservices composition requires engineers to have a view of the behavior of the various components at runtime. Given the potential for harm that can arise from persistent attacks by hostile entities at runtime, there is a need to monitor microservices and their runtime environments (Haselböck & Weinreich, 2017, Peinl, 2016). Monitoring allows identification, detection, and even ability to foresee critical events and situations that occur during runtime (Gander et al., 2013, Asim et al., 2014).
6. *Securely respond to attacks using adaptation mechanisms (SDA-6)*- microservices compositions are distributed systems implemented using a collection of components that independently evolve and react to their environments and other external factors. The distributed nature of component creates many potential points of failures. A microservices composition should be built to withstand failures of individual components. Mechanisms that ensure that the application responds adequately to changes at runtime to maintain an appropriate security posture are referred to as secure adaptation mechanisms (Gabbrielli et al., 2016, Florio et al., 2016, Hassan & Bahsoon, 2016).

The six activities listed above can be view as the development activities for secure microservices compositions. Next, a conclusion is provided.

5.6 Conclusion

Nowadays, the impact of security breaches on an enterprise can be overwhelming. Microservices compositions, like any web applications should therefore be developed with security in mind. Building secure microservices compositions is, however, a complicated exercise. This chapter has laid the foundation towards understanding the microservices architecture's security threats using an attacker's point of view and identified security requirements common to most implementation of the microservice architecture. Suitable protection measures have also been identified. Even when security requirements and protection measures are identified early during the development process, there is still the challenge of validating if the implementation of various components of the architecture is safe. Furthermore, there is a need to ensure that the runtime environment does not provide attackers with the means to control or compromise the microservices composition. To this end, this chapter identified security-focused activities that ensure that the suitable protection measures are implemented.

Although building a secure microservices composition is a complex exercise, the identification of security-focused activities in this chapter can go a long way towards achieving the security goals of a microservices composition. The security-focused activities include documenting security requirements at design time to provide a security strategy to secure various assets of the microservices composition. Furthermore, this chapter also noted the importance of encouraging microservices software engineers to adopt documented security guidelines to eliminate known security vulnerabilities in microservices compositions. Also, ensuring comprehensive security testing is another vital activity required to identify and eliminate security flaws before microservices compositions are deployed to a production environment. The chapter has also identified the importance of creating a secure runtime infrastructure for microservices compositions. In addition, mechanisms should be in place to ensure that software engineers have insight into the behavior of microservices at all times to be able to identify attacks. If a microservice composition is attacked, the composition should be equipped with the ability to respond and maintain its security posture.

In this regard, this chapter provided a necessary preliminary security risk analysis of the entire composition by applying a threat modeling technique to identify security threats in a microservices composition. Then, the various security flaws often associated with each threat were identified and used to derive various security requirements common to most microservices compositions. Furthermore, the suggested protection measures were used to derive a set of important security-focused activities that should be incorporated into the development process of a microservices composition. The activities constitute the secure development activities of a microservices composition.

The next chapter reviews the literature on the six identified security-focused activities that form the secure development activities of a microservices composition. The aim of the review is to identify various tools, techniques, and methods that can be used to support software engineers in incorporating the six secure development activities to become part of their daily software development task.

Chapter 6

Secure Microservices Development

6. 0 Introduction

The threat modeling exercise in Chapter 5 identified six software development activities to consider so that the security goals of a microservices composition can be achieved. The challenge in a microservices development environment is that microservices are typically developed using the Agile methodology, where the general perception is that the incorporation of security in the development process would be against agile values (Oyetoyan et al., 2016). Consequently, many Agile teams tend to release software without performing full-scale security testing, and later address any security vulnerabilities by deploying new releases (Heinrich et al., 2017). The general perception of security within Agile teams has had a significant impact on the development of software in general. The general reluctance is reflected in the State of Software Security report of 2017 were the top ten common security vulnerabilities in 2017 were strikingly similar to that of 2016 (Veracode, 2017). Finding ways to seamlessly incorporate the six software development activities in a manner that does not impact the Agile values and benefits will thus be beneficial to the microservices development community.

This chapter aims to understand how to effectively incorporate the six secure development activities using existing tools, techniques, and methods so that the security of microservices compositions can be improved. The organization of this chapter is as follows. First Section 6.1 introduces the approach that is used to identify and review the relevant literature. Section 6.2 identifies essential concepts that assist in documenting security activities of microservices compositions effectively. Section 6.3 obtains secure programming practices that can ensure that microservices are designed and developed in a manner that makes them inherently safe. Section 6.4 identifies and reviews available tools that can be used to determine security flaws during the development process of a microservices-based application. Section

6.5 identifies tools that can be used to create safe runtime environments for microservices to prevent the environment from being a source of security risks to the application. Next, Section 6.6 identifies and reviews mechanisms that can give continuous insight into the behavior and health status of microservices to detect attacks at runtime. Section 6.7 identifies mechanisms that can be used to enable microservices to react to malicious attacks at runtime securely. A summary and conclusion follow in Section 6.8 and 6.9 respectively.

6. 1 Review of software development activities for secure microservices

The microservices architecture can be considered a relatively new area of research (Di Francesco, Malavolta & Lago, 2017). In this regard, this chapter adopts a systematic mapping research approach (Petersen et al., 2008) that is commonly used for research areas that are not yet mature (Kitchenham & Charters, 2007). The process used in this chapter is first to formulate guiding research questions on each software development activity for secure microservices compositions identified in Chapter 5. Each question is used as criteria to identify relevant literature from publications. An analysis is performed for each activity to identify potential areas that still require research attention.

Most security challenges experienced in a microservices architecture are those generally familiar to all SOA implementations (Dragoni et al., 2016). Any new microservices security problem that is not found in general SOA implementations can be due to network complexity arising from the arrangement of components within the microservices architecture. The review in this chapter, therefore, considers not only literature specific to microservices but also literature relevant to SOA in general. Journals, conference publications, and various industry articles are reviewed.

The structure of the discussion to follow is according to the identified secure software development activities for secure microservices compositions namely:

- Document security requirements of microservices compositions
- Adopt secure programming best practices
- Validate security requirements and secure coding standards
- Secure configuration of runtime infrastructure
- Continuously monitor components of the microservices composition
- Securely respond to attacks using adaptation mechanisms

Next, the documentation of the security requirements of microservices compositions is reviewed.

6.2 Document security requirements of microservices compositions

A secure microservices composition can be created by implementing the security services dictated by ISO 7498-2 namely authentication, access control, data confidentiality, data integrity, and non-repudiation. A comprehensive specification of protections mechanisms can support effective authentication, access control, data confidentiality, data integrity, and non-repudiation in a microservices composition (El Hassani et al., 2015). A security policy is used to specify protection mechanisms in order to manage security (ISO, 1989). Consequently, an understanding of different types of security policies applicable to microservices becomes essential to specify comprehensive protection measures effectively. To this end, Table 6.1 formulates questions to ensure effective management of the security concerns of a microservices composition. A prefix REVQ denotes each question to differentiate from the research questions formulated in Chapter 1.

Table 6.1. Guiding questions on microservices security policies

Research questions	Motivation
REVQ1. How does the microservices architecture affect the design of security policies for a microservices composition?	To identify the attributes of a good microservices security policy design.
REVQ2. What types of security policies are required to document protection measures for a microservices composition comprehensively?	To gain a comprehensive understanding of the different security policy types required in a microservices composition.

To be able to answer REVQ1 from Table 6.1, the next section identifies characteristic of the microservices architecture that affects the design of security policies.

6.2.1 Microservices architecture and security policies

In Chapter 3, microservices, the API gateway, the services registry, the message broker and containers were identified as critical assets of the microservices architecture. These assets are collectively referred to as *components* of the microservices composition. A security policy needs to consider that components of a microservices composition are loosely coupled and distributed, where each component may have unique security needs. The following two essential attributes should be taken into account:

- *Support for distributed and loosely coupled microservices interactions* (Brodecki, Szychowiak & Sasak, 2012, Suzic et al., 2016, Azarmi & Bhargava, 2017). In the composition, some components need to delegate access control decisions to other components because not every component can directly ask the end-user for authentication details (Nacer et al., 2017). In PickMeUp, the *Driver Management Microservice* relies on the *Trip Management Microservice* and the *Passenger Management Service* to ensure that the passenger requesting a trip is a valid requestor of information. The security policy defined in such a scenario should determine protection mechanisms to enable a trust relationship between collaborating microservices. Furthermore, since collaboration between microservices is predominantly defined over Hyper Text Transfer Protocol (HTTP), the security policy should also define protection mechanisms to support strong transport-layer security required for secure communication between distributed components of the microservices architecture.
- *Support for hierarchical security policy domains* (Dell'Amico et al., 2013*).* Microservices compositions require a high-level security policy to define protection mechanisms for the entire composition, and also security policies specific to each component of the composition. Each component of the composition represents a domain where component-specific security policies can be enforced. To this end, the implementation of the architecture represents a hierarchy of domains where policies are defined and enforced. The hierarchy ensures that certain sensitive information remains confined to one component such as a particular microservice and some information is securely shared between collaborating components. The hierarchical structure thus requires security policy language support.

To be able to answer REVQ2 from Table 6.1, the next section identifies the various types of security policies relevant in a microservices composition.

6.2.2 Categories of security policies

In Chapter 5, the threat modeling of PickMeUp suggested protection mechanisms to mitigate threats using the STRIDE threat classification model. From the threat analysis, the following list of policies may be able to document the suggested protection mechanisms sufficiently:

1. *Data protection policy* (Satoh & Tokuda, 2015). Threats related to spoofing, data tampering, and information disclosure require a data protection policy to protect the data in a composition. The policy should define an encryption schema to protect data in transit and storage.
2. *Access control policy* (Andrews, Steinau & Reichert, 2017). An access control policy is required to define permissions assigned to subjects that access components of the microservices composition. Subjects in the context of a microservices composition include end-users, and also components of the composition that require access to other components to function. Access control aims to limit the damage when a component of the composition is compromised by defining permissions required by each component and resources accessed by each component.
3. *Microservice technology-specific policy* (Yu et al., 2018). Microservices are built using software libraries that may contain security vulnerabilities. In the 2017 State of Software Security report, eighty-eight percent (88%) of applications written in Java, a popular technology for creating microservices, had at least one component-based vulnerability (Veracode, 2017). Therefore, microservices should have a technology-specific security policy focusing on mitigating weaknesses in libraries. For example, for microservices developed using Java technology, a technology-specific security policy should focus among others on mitigating the effects of the OWASP Java top ten vulnerabilities (OWASP, 2013).
4. *Network security policy* (Yu et al., 2018). The network security policy is vital to control access to components of the microservice composition effectively. For example, In the PickMeUp, the *Trip Management Microservice* does not need direct access to passenger and driver information. As a result, the network security policy should deny connection attempts from the *Trip Management Microservice* to a passenger database. In the event of *Trip Management Microservice* being compromised, the attacker will not have direct access to critical data. The network policy can also be used to specify how logical addresses are allocated, distributed, and managed for containers or virtual machines
5. *Microservices composition security policies* (Satoh & Tokuda, 2015). The security policy of the entire microservices composition can be viewed as a combination of a policy to protect messages exchanges between components of the microservices composition and an access control policy defining how components are accessed. The composition security policy should describe how exchanged messages are protected to ensure integrity and confidentiality. The policy should also restrict access to specific capabilities provided by microservices. For example, in PickMeUp the

security policy may ensure that *Passenger Notification Microservices* is only allowed to call actions on the *Passenger Management Microservice* to query details of a specific passenger but may not update records on the database.

6. *Virtual Machine and container security policies.* The impact of security vulnerabilities in virtual machine or containers cannot be ignored. Over thirty percent (30%) of official images in docker hub, for example, contain high priority vulnerabilities (Yu et al., 2018). A security policy is therefore required to mitigate against security vulnerabilities that may exist in the runtime environments. The virtual machine or container security policy should define mechanisms to prevent container breaches and in the event of breaches limit damages that can occur. For example, the policy may determine mechanisms that block components of the microservices composition from reading unsafe directories and ensure that each component is granted least privileges to perform a function.

6.2.3 Challenges of a microservices composition security policy

Specifying a security policy for a microservices composition is a complex exercise. To begin with, there are no policy languages and a standard to specify a security policy for RESTful web services (Yu et al., 2018). Moreover, when the microservices composition security policy is created by combining the security policies of each component, software engineers are forced to study multiple security policy representations without tool support to identify possible security policy inconsistencies. The task is further made complex because there is no precise definition of what a security policy inconsistency is in such compositions (Satoh & Tokuda, 2011, Yu et al., 2018). Furthermore, a microservices composition may compose other microservices compositions recursively resulting in security policies being applied recursively. In such scenarios, a software engineer has to check the security policy hierarchy to confirm that appropriate security is applied end-to-end, and this can be a daunting task.

6.2.4 General observation and discussion

The unavailability of established policy specification languages to represent security policies for RESTful web services requires that a high-level language be used to manage microservices security concerns. The absence of a common policy specification language means that there is no standardized way to communication security requirements across different teams developing microservices. This presents a security challenge when microservices from different teams collaborate to automate a business function. engineers of microservices should, therefore, ensure that the APIs of all microservices treat all input values as untrusted data that require strict validation. Despite the non-availability of a common security policy specification language for RESTful services, existing security policy models such as access control models can still be applied by each component of the microservices composition.

In addition to managing functional security requirements of microservices, the security of microservices can be enhanced by using a collection of procedures or suggestions for best practices within an organization. Secure programming best practices are part of procedures suggested in Chapter 5 that assist in developing secure microservices compositions. The next section identifies appropriate programmable practices and investigates how to use these practices to create safe microservices.

6.3 Adopt secure programming best practices

Microservices compositions, like any traditional web application, are not immune to known security attacks such as SQL injections and Cross Site Scripting (XSS) (Ahmadvand & Ibrahim 2016). Each year, most of the top ten web application vulnerabilities published by OWASP, (OWASP, 2017) are vulnerabilities with documented guidelines on how to prevent attacks. Furthermore, in the 2017 State of the Software Security report (Veracode, 2017), thirty percent of the web applications surveyed still had SQL injection vulnerabilities. The recurring attacks are a result of software engineers not adhering to documented security guidelines (Aljawarneh, Alawneh & Jaradat, 2017). Designing and developing secure microservices compositions should, therefore, prioritize safe programming practices (Zhu et al., 2014). This section aims to understand how secure programming practices can assist develop safe microservices. The questions in Table 6.2 below provide a guideline to the review.

Table 6.2. Guiding questions on secure programming practices

Research questions	**Motivation**
REVQ3. Which secure programming best practices can assist engineers to avoid known security flaws when developing microservices?	The aim is to elicit a list of secure programming practices that can help eliminate security flaws in microservices compositions.
REVQ4. How can software engineers adopt secure programming practices without affecting the rate of microservices releases?	The aim is to identify ways to enforce secure programming practices without impacting Agile values.

The next section first identifies a taxonomy to assist in eliciting relevant secure programming best practices, and then use the taxonomy to identify relevant secure programming best practices.

6.3.1 Taxonomy of secure programming practices

The Microsoft Secure Development Lifecycle defines four security principles namely secure-by-design, secure-by-default, secure-by-deployment, and secure-by-communication, that provide a basis to reason about secure software development in general (Howard & Lipner, 2006). This section uses the four security principles as a taxonomy to assist in eliciting secure programming practices. First, the principles are defined below in the context of a microservices composition.

- *Secure-by-design* - the architecture of a composition should contain a security discussion detailing how security risks are mitigated, and how the attack surface is minimized. The premise of secure-by-design is that it is often difficult for a microservices composition that is designed in an insecure manner to be made secure after implementation and deployment. Secure-by-design implies that components of a microservices composition are designed to be inherently secure.
- *Secure-by-default* - all components of a microservices composition should be provided with a default configuration that is secure. Any insecure runtime configuration should be a result of a deliberate effort by the user. For example, in PickMeUp, enabling default password aging and password complexity for end users and ensuring that each microservice is deployed with a least set of privileges goes a long way towards ensuring secure-by-default.
- *Secure-by-deployment* - the deployment pipeline for microservices is safe. For example, the PickMeUp deployment pipeline which may constitute the source code version control system, the tools for continuous integration, and mechanisms used to deploy the final artefacts on containers should all be safe.
- *Secure-by-communication* - software engineers should respond promptly to reported security vulnerabilities. The expectation is that software engineers are aware of newly reported vulnerabilities and engineers can identify the potential impact any new vulnerabilities may have on the microservices composition.

The next section attempts to address REVQ3 from Table 6.2 by using the four security principles discussed above to identify secure programming best practices relevant to a microservices composition.

(a) Secure programming practices for secure-by-design

A security flaw in the design of a microservices composition may result in a security breach (Arce et al., 2014, Williams & Woodward, 2015). In this regard, security requirements should be incorporated into the initial architectural design of microservices compositions (Kadam & Joshi, 2015, Athanasopoulos et al., 2015). Furthermore, the following guidelines are vital for safe microservices designs:

- *Keep microservices design simple*. Complex designs increase the likelihood that errors are made in implementation, configuration and use of microservices. Furthermore, an elaborate design makes it hard to enforce security policies (Sahu & Tomar, 2017).
- *Ensure input validation on the microservices API.* Microservices should ensure that all input data from untrusted data sources is validated. Validating input data eliminates a majority of injection vulnerabilities (Almorsy, Grundy & Müller, 2016, Sahu & Tomar, 2017).
- *Give attention to source code compiler warnings.* Microservices source code should be compiled using the highest source code compiler warning level. Any compiler warnings should be eliminated by modifying the code (White, 2015).
- *Sanitize data sent to other microservices.* Sanitization is cleaning or filtering input data. In microservices, this involves checking for invalid UTF-8 encoding, removing line breaks, tabs and extra white space and stripping octets in the input. The output of each microservice should be secured by stripping out unwanted data. Sanitizing data helps secure data before rendering to end user and prevents cross-site scripting attacks.

(b) Secure programming practices for secure-by-default

Ideally, microservices compositions should be inherently safe on deployment (Stanek, 2017). Microservices compositions can be developed in a manner that makes them inherently safe when engineers adopt the following practices:

- *Adhering to the principle of least privilege.* Components of a microservices composition should execute with the least set of permission required to perform a function (Sittig & Singh, 2016, Neumann, 2018). For example, a microservice should not have access to directories, databases tables, and any other resources that are not required to perform its function. Any elevation in permissions or privileges when needed should be authorized and held for a minimum period.
- *Practicing defense-in-depth.* Security risks in a microservices composition can be managed by using multiple defense strategies. When one layer of defense becomes inadequate, another layer of protection can be relied upon to prevent an attack (Williams & McCauley, 2016, Gkioulos &

Wolthusen, 2017). For example, a composition should rely on transport-layer security to secure a connection between microservices as the first line of defense and microservices should use access tokens to authenticate each other as the second line of defense in a REST communication model.

- *Denying access by default.* Denying access by default means that by default, access to components of the microservices composition should be rejected, and a protection scheme should define conditions when access is permitted (Bertolino et al., 2014, Ulltveit-Moe & Oleshchuk, 2015).

(c) Secure programming practices for secure-by-deployment

The deployment process of microservices can provide a path that an attacker can exploit to make harmful changes and deploy such changes into production environments or even perform a denial of service attack. Secure-by-deployment requires the continuous delivery toolchain, the build, and test environment to be secured so that changes are safely made in a repeatable and traceable manner. The following best practices can go a long way towards ensuring secure deployments:

- *Guaranteeing safe deployment pipelines.* The deployment pipeline of microservices is made up of the source code version control system, the continuous integration tool for compiling source code, scripts of transferring the final artefacts to the production server, and mechanisms for provisioning Docker containers on the production server. Secure by deployment requires that each step in the deployment process be safe to prevent breaches (Gruhn, Hannebauer & John, 2013, Bass et al., 2015). The deployment pipeline should also provide trusted components that mediate access to sensitive configuration information required during deployments.

(d) Secure by communication in microservices

Secure-by-communication requires that development teams respond promptly to reports of security vulnerabilities and communicate information about security updates. To this end, engineers need to keep abreast with new security vulnerabilities and have access to the latest security information. Software engineers should, therefore, continuously acquire new information on most recent security exploits.

Although secure programming practices discussed above can adequately ensure safe microservices, the theoretical knowledge of secure programming practices is not enough. The challenge as outlined in a study by Veracode (2011) is a reluctance by software engineers to adopt secure programming practices. A survey by Oyetoyan et al. (2016) noted that secure programming was practiced by less than fifty percent (50%) of engineers and forty percent (40%) of architects. There is, therefore, a need to find ways to encourage the adoption of secure programming practices. To this end, the next section attempts to address REVQ4 from Table 6.2 by identifying ways to promote the adoption of secure programming practices.

6.3.2 Enforcing secure programming practices

Zhu et al. (2013) attribute the reluctance to adopt secure programming practices to the fact that only a small fraction of engineers is well-trained in secure software development. Efforts to encourage the adoption of secure programming practices should therefore not underestimate the fact that many engineers may not be trained in security. To this end, suitable methods to integrate and enforce secure programming practices are those that seamlessly incorporate the practices into the day-to-day software development activities of engineers in such a way that engineers view security as part of the expected behavior of the microservices composition. Table 6.3 below consider each programming practice and suggest methods to incorporate the programming practice into the microservices development process seamlessly.

Table 6.3. Methods to enforce secure programming practices

Security principles	**Secure programming practices**	**Methods to enforce the practices**
1. Secure-by-design	Keep the microservices design simple	• Document safe coding standards and ensure reviews of designs and source code • On developer's integrated development environment (IDE) and continuous integration tools use an analysis tool to detect any cyclomatic complexity of the microservices source code
	Ensure input validation on the microservices API	• Create security unit tests for input validations as part of the existing unit testing framework. The test suite should consider the use and misuse cases to microservices API

Security principles	Secure programming practices	Methods to enforce the practices
1. Secure-by-design (continued)	Give attention to compiler warnings	• Install compiler warning plugins on continuous integration tool and ensure those compiler warnings are set to the highest level
	Sanitize data sent to other microservices	• Create security unit tests that ensure that microservices responses do not contain invalid data and also assert that errors and exceptions are caught in a manner that makes microservices secure
2. Secure-by-default	Adhering to the principle of least privilege	• Ensure non-essentials operating system services are disabled • Automate the creation of Docker containers or virtual machines and their security configurations and ensure security configurations are comprehensively validated • Microservices should always use HTTPS • For Unix-based systems disable root login
	Practicing defense in depth	• Perform penetration testing
	Denying access by default	• Write security unit tests that assert that microservices have no access to resources they should not be permitted to access • Ensure all microservices administration interfaces are protected
3. Secure by deployment	Ensuring secure deployment pipelines	• Document secure deployment guidelines • Ensure secure access to microservices source code and continuous delivery toolchain and ensure binaries and other build artefacts are signed to prevent tampering • Ensure that keys, secrets, and credentials are not stored in source code or plain text but in a secure secret management system • Monitor the continuous integration and continuous delivery servers
4.Secure-by-communication		• Ensure that the development team has access to the latest security news and reports

6.3.3 General observation and discussion

The suggested methods to enforce secure programming practices in Table 6.3 provide a practice-oriented effort to incorporate secure programming practices in the development of secure microservices compositions. The suggested methods require a bold initiative in teams that use Agile methods and continuous delivery practices because the methods at the beginning are likely to have a negative impact on the frequency of software releases, and require engineers to change their perception about security. The following is vital to ensure seamless and effective adoption of secure programming practices:

- Translate secure programming practices into security requirements of a microservices composition so that software developer view security as part of the expected behavior of the composition. In this way, engineers will prioritize the security of microservices compositions.
- Use the continuous integration and continuous delivery toolchain to automate the identification of violations of secure programming practices and security requirements. The toolchain can be relied upon to provide early security testing feedback to engineers so that the security flaws are addressed before microservices are deployed to any environment.

Although using secure programming practices is vital in the development of microservices as discussed above, the benefits of secure programming practices can only be realized when security testing is treated as an essential step in the microservices development process. The next section discusses how security testing can be incorporated into the development process as an essential security-focused activity for secure microservices.

6.4 Validate security requirements and secure programming best practices

In the context microservices, security testing can broadly be defined as testing requirements of microservices that concern to data confidentiality, data integrity, authentication, authorization, non-repudiation and also validating the ability of the microservices runtime environment to withstand attacks (Paul, 2016). Security testing generally validates the correct implementation of specified security requirements and identifies unintended vulnerabilities, by mostly using penetration testing attempts or simulated attacks (Tian-yang, Yin-Sheng & You-yuan, 2010). In a fast-paced development environment such as microservices, it makes sense to automate the validation of security requirements and secure programming best practices. To this end, this section first identifies suitable criteria to evaluate if available security-focused tools can be used for automated testing of microservices. The identified criteria are used to review the feasibility of automating available tools for validating security requirements and secure programming best practices. Questions in Table 6.4 are used to guide the review.

Table 6.4. Formulated research questions on automated security testing

Research questions	**Motivation**
REVQ5. What attributes should a security-oriented testing tool possess to seamlessly integrate into a fast-paced microservices development environment?	The aim is to identify criteria to use to evaluate available tool and determine the feasibility of automating the tools in the microservices development process so that security is given early attention in microservices.
REVQ6. Can existing security testing tools be used to automate security testing in microservices?	The aim is to identify tools that are candidates for security testing automation so that less human effort is used on security testing.

The next section attempts to answer question REVQ5 from Table 6.4 by identifying the attributes required of security testing tools.

6.4.1 Required attributes of security testing tools

The effective use of a tool for automated security testing in microservices depends on how seamlessly the tools integrate into the microservices development process. In fast-paced development environments such as microservices, a tool with the following attributes is likely to seamlessly integrate into the development process without negatively impacting software engineers' productivity:

- *Ease to integrate* (Kaur, 2017). Similar to any tool used to analyze source code, security testing tools should be easy to integrate into an IDE or a continuous integration tool. Ease of integration increases the chances of the tool being adopted without impacting a software developer's productivity.
- *Easy to use (*Le Ru, 2015*).* An ideal testing tool should not require engineers to have advanced security knowledge to use. As indicated by Zhu et al., (2013), many software engineers are generally not trained in security.
- *Natural results interpretation (de Andrade Gomes et al., 2017).* engineers should effortlessly be able to understand reported security flaws, and if possible, the tool should provide details on how the engineers can fix the security flaws.
- *Extensive language support and portability.* The microservices architecture is technology agnostic. The tool should, therefore, not limit engineers to a particular programming language or development platforms.
- *Extensibility.* The tool should allow engineers to add new capability when new security vulnerabilities are reported.

Various security testing tools are readily available (Kuusela, 2017). The challenge in a microservices development process is to identify which tools to integrate to validate security requirements. The next section uses the five attributes discussed above to review readily available tools and identify their suitability for automation and answer REVQ6 in Table 6.4.

6.4.2 Review of security testing tools

The integration of security testing tools in software development processes is an opportunity which many software engineers has not fully exploited (Peischl, Felderer & Beer, 2016). There is a general ignorance of security testing in many development teams and a significant dependency on external vendors to perform penetration testing (Cruzes et al., 2017). Consequently, in many software development teams, little attention is paid to security testing (Shuaibu et al., 2015). In order to address the security testing challenge, this section reviews readily available tools to identify which tools are suitable for automated security testing in microservices. First, the types of security testing are defined:

- *Static security testing* – checks the source code, design documents to find errors, code flaws, and potentially malicious code when the code is not being executed.
- *Dynamic security testing* - validates the runtime behavior of the application for security mechanisms when source code is being executed.

Table 6.5 reviews the popular, readily available, and non-proprietary tools namely GauntIt (Kuusela, 2017), SonarQube (Campbell & Papapetrou, 2013), and FindSecurityBugs (Kuusela, 2017), for their suitability in microservices using the attributes discussed in section 6.4.1 above. A comprehensive list of both commercial and non-commercial tools is reviewed by Kuusela (2017).

Table 6.5. Review of security testing tools

	SonarQube	**Gauntlt**	**FindSecurityBug**
Description	SonarQube is an open source tool for inspection of code quality (Campbell & Papapetrou, 2013)	Gauntlt is a security testing framework that incorporates other security testing tool (Kuusela, 2017)	FindSecurityBugs is an open source security testing tool (Kuusela, 2017)
Testing approach	Both static and dynamic functional testing (SonarQube, 2018).	Dynamic testing functional testing (Gauntlt, 2017)	Static testing functional testing
Ease to integrate	Easy set up with simplified integration using dedicated plugins for IDEs and continuous integration tool	Gauntlt is easy to integrate with continuous delivery tools and other testing tools. Installing attack tools and maintaining them may take time	Easy integration into with continuous delivery tools and plugins exist for many IDEs
Easy to use	Easy to use on IDE or using a browser interface. A software engineer can specify their quality gates	Security-attack scenarios are described in a straightforward human-readable language called Cucumber (Gauntlt, 2017).	FindSecurityBugs is simple to use, and there are plugins to integrate into an IDE.
Results presentation	Results displayed on a user interface with notes on detected issues	Gauntlt require custom parser to automate results to desired granularity	Results exportable into a format understood by continuous delivery tools
Portability	Provides support for more than 20 programming languages like Java, C++, and C#, etc (SonarQube, 2018)	Gauntlt requires Ruby to be installed (Storms, 2015)	FindSecurityBugs support only for Java web applications
Extensibility	An extension guide is provided to assign engineers to write their plugins (SonarQube, 2018)	New attack scenarios can be written using a language called cucumber (Storms, 2015)	FindSecurityBugs can be extended by writing new detectors

6.4.3 General Observations and Discussion

Table 6.5 above indicates that tools such as SonarQube, GauntIt, and FindSecurityBugs can easily be integrated into software engineer's IDEs and continuous integration tools, and are easy to use and extensible. The tools are all suitable for automating security testing in microservices development environments. Among the reviewed tools, SonarQube is more suited because it supports many languages and frameworks, provides both static and dynamic testing, and is easy to use and can also be integrated to other tools like FindSecurityBugs.

The successful use of security testing tools as discussed above, however, requires upfront effort to ensure that a baseline of ready-to-use security tests cases is provided. The security test cases should preferably be written in a manner that is understandable by all stakeholders to ensure broader collaboration. Software development teams should, therefore, consider incorporating a development methodology that supports communication between the business customers, engineers, and the testers, by using acceptance tests cases commonly used in behavioral-driven development (Solis & Wang, 2011). Furthermore, the use of security testing tools should be informed by the results of a thorough threat modeling process as discussed in Chapter 5. The threat modeling techniques and the available tools discussed above provide a platform for providing innovative techniques to promote comprehensive security testing (Peischl, Felderer & Beer, 2016).

In addition to security testing discussed in the section above, the threat modeling exercise conducted in Chapter 5 identified securing the runtime environment as one of the methods that can reduce the attack surface of microservices compositions. The next section reviews mechanisms to ensure secure runtime environments of microservices.

6.5 Secure configuration of runtime infrastructure

Most types of attacks on microservices, whether executed through the network channel or on the composition, ultimately target the runtime environments where data is stored, and the microservices run. Any runtime environment that allows default user accounts, run unnecessary operating system services, and unpatched software libraries provide an attacker with pathways to gain control of microservices (Terala & Cole, 2015). The runtime should, therefore, be provisioned in a manner that ensures the safety of microservices. The first step towards ensuring a secure runtime environment is to formalize a secure configuration baseline of both hardware and software components, and then later validate the configuration when microservices are deployed (Hochstein & Moser, 2017).

Currently, various tools are used in industry to configure the runtime environments albeit not for security (Robinson & Northcut, 2016). It makes sense to extend the use of available tools so that the provisioning of the runtime infrastructure also integrates with security aspects. This section aims to understand if it is possible to create a re-usable template-based security configuration that automatically scales and support rapid deployment of secure microservices using available tools. Questions in Table 6.6 below are used to guide the evaluation.

Table 6.6. Formulated research questions on secure configurations

Research questions	**Motivation**
REVQ7. What capabilities makes a tool suitable for creating secure runtime environments as part of the microservices development process?	To identify criteria that can be used to identify tools that can be effectively used to create secure microservices runtime environments.
REVQ8. Can the widely used configuration management tools be easily used to create secure microservices runtime environment?	To understand how to leverage the available tools to create secure microservices runtime environments.

A vital step towards evaluating the feasibility of using the available tools for automating the creation of secure runtime infrastructure for microservices is first to identify an evaluation criterion suitable for microservices. The next section identifies the criteria and addresses REVQ7 in Table 6.6.

6.5.1 Capabilities for secure configurations

The following capabilities are vital to ensure the effective creation of secure runtime environments for microservices:

- *Support for different security requirements* (Tang et al., 2015). The microservices composition is a distributed system composed of components that may have different security needs. As a result, each component should have its security configuration that defines the security concerns of the component (Torberntsson & Rydin, 2014, Wettinger et al., 2014).
- *Allow authoring and version control of security configuration files (Morris, 2016,* Ikeshita et al., 2017*).* In a fast-paced microservices development environment, it is essential to ensure that security configurations files are treated like software source code so that any changes on the configuration files are tracked and go through the formal change control process before deployment.
- *Support for validation of configurations* (Huang et al., 2015, Baset et al., 2017). Any changes to configurations should be tested first to ensure configurations are not a source of vulnerabilities.

- *Easy management of dependencies between configuration files* (Tang et al., 2015). Dependency between security configuration files may become unavoidable. These dependencies should preferably be expressed the same way as software source code dependencies for easier maintenance.
- *Low learning curve* (Fernández et al., 2016). Security configuration files should be written in languages that are close to natural language to make it easy for the software engineers with less security knowledge to be able to maintain (Fernández et al., 2016).

The next section evaluates the feasibility of leveraging the widely used tools to create re-usable template-based security configurations. Characteristics identified above are used in the evaluation. The section aims to address question REVQ8 from Table 6.6.

6.5.2 Review of tools for secure configurations

The tools that are common in the industry for configuration management are Chef (Taylor & Vargo 2014), Puppet (Loope, 2011), Ansible (Hall, 2015), and SaltStack (Myers, 2016). Table 6.7 below provides a review of each tools using the capabilities discussed above.

Table 6.7. Review of configuration management tools

	Chef	**Ansible**	**SaltStack**	**Puppet**
Description	Chef is an open source configuration management tool. Chef uses a pure-Ruby domain-specific language (DSL) for describing the state of system resources, packages to be installed, services that should be running (Taylor & Vargo, 20141)	Ansible is a tool for automating software provisioning and configuration management. Configuration files are written in YAML format. The YAML file represents a task also known as a play and stored in playbooks (Hall, 2015).	SaltStack is an open source configuration management tool. It is written in Python and is designed to be highly modular and easily extensible (Myers, 2016).	Puppet is also an open configuration management tool. The tool follows the concept of agent- master relationship (Loope, 2011). State of system resources is described using a declarative language or a Ruby DSL and stored in files called Puppet manifests. Compiled manifests reapplied on a target system.

	Chef	**Ansible**	**SaltStack**	**Puppet**
Platform Support	Chef is supported on Linux, Mac OS, and Microsoft Windows platforms. Chef integrates into cloud-based platforms such as Amazon EC2 and Microsoft Azure (Marschall, 2015).	Ansible is supported on Linux. Mac Os, Solaris. On Microsoft Windows platforms Ansible requires a Linux control machine (Hall, 2015).	SaltStack is supported on Linux, Mac Os, Microsoft Windows platforms (Hall, 2016). It is not open source and cannot be customized easily.	Puppet runs on Linux, Mac Os, Microsoft Windows platforms (Loope, 2011).
Support for diverse use cases	Different configurations called recipes can be written using Ruby for each use case. The recipes can be grouped to form a cookbook for more natural management. More than 800 recipes are freely available (Marschall, 2015).	Different playbooks can be used for each use case (Hall, 2015). The Ansible's architecture is based on controlling machines and nodes. However, more features are in a paid version.	The DSL used by SaltStack provide a rich set of features for a diverse use case (Hall, 2016).	Different resources can be used for each use case. Puppet main features are in its paid enterprise version (Loope, 2011).
Authoring and version control	Recipes can be versioned and stored in version control (Marschall, 2015).	Ansible playbooks can be stored in version control systems (Hall, 2013).	Recent versions of SaltStack support integration to version control.	Puppet configuration files can be stored in version control systems.
Validation	Chef supports unit tests, functional tests, and integration tests of recipes (Marschall, 2015).	Ansible support unit test, functional tests, and integration tests of playbooks (Hall, 2013).	SaltStack provides a unit test and integration test suite.	Puppet support unit tests, functional tests, and integration tests of resources (Loope, 2011).

	Chef	**Ansible**	**SaltStack**	**Puppet**
Dependency management	Dependencies between recipes can be defined on each configuration file (Marschall, 2015).	A playbook can import or reference other playbooks (Hall, 2013).	All of the SaltStack execution modules are available to each other and modules can call functions available in other execution modules (Hall, 2016).	Puppet maintains a graphical representation of the list of resources and their interdependencies (Loope, 2011).
Easy learning	Chef is easy to install and set-up. However, recipes require knowledge of Ruby to write or understand. Chef has an active support community. (Marschall, 2015). The documentation although rich concerning content can become difficult to read because the user has to navigate through many links on the website.	Ansible is easy to install and set-up. The playbooks are expressed in YAML format that has minimum syntax and is easy to learn because it is close to simple English representation.	SaltStack is difficult to set-up for new users as the available documentation is difficult to understand at the introductory level (Tsumak, 2016). The web UI is newer and less complete compared to other tools. Support for non-Linux platform is not good.	Puppet provides simple installation and initial setup. The user interface is mature and complete. There is a well-established support community. Puppet also has a robust reporting capability (Loope, 2011).

6.5.3 General observations and discussion

The review of existing configuration management tools in Table 6.7 above shows that the available tools possess the required capabilities to create reusable template-based security configurations. The available configuration management tools are primarily similar in functionality, and engineers of microservices should be able to use any tool of their choice. In a development team that has no previous experience with any tool, choosing an ideal tool may be hard. In order to assist software engineers to select an ideal tool, the review identifies the following critical points for each tool:

- Chef is a software engineer-friendly platform and provides various tools for engineers such as the Chef developer toolkit and the Chef knife plugin. Chef offers many different recipes or modules for free that engineers can use off-the-shelf. The capability to create different recipes using Ruby makes Chef a highly customizable configuration management tool. However, Chef had three reported security vulnerabilities of medium severity listed on the common vulnerability exposure (CVE) database at the time of writing, and engineers should ensure that an attacker does not exploit these.

- Puppet is considered a system administrator-oriented tool and may be difficult to use for a software engineers with no system administration background. However, previous knowledge of system administration is becoming less relevant with each new release as the tool is improved. Most of the functionality of Puppet is in the paid-for enterprise version. A development team that does not need many features can undoubtedly use the open source version. However, the team may need engineers with strong Ruby programming skills to add new capabilities when the need arises. Puppet had fifty-one vulnerabilities on CVE at the time of writing.
- Ansible is relatively new and is therefore not as mature compared to Chef and Puppet. However, Ansible is a better choice for new and small development teams that do not want to spend much on configuration tools and do not have the time to learn how to use some of the more complicated configuration management tools. The drawback of Ansible is that the tool is not easy to customize because it is not open source and the graphical user interface has insufficient features. Ansible had six security vulnerabilities on CVE at the time of writing.
- SaltStack's most significant advantage is scalability and resiliency. SaltStack provides multiple levels of masters in a tiered arrangement that both distributes load and increases redundancy. SaltStack, however, lacks many features and refinements to make the tool enterprise-ready. The tools lack support for strict transport-layer security. The tool also poses a significant learning curve for new users. SaltStack had twenty-one security vulnerabilities on CVE at the time of writing.

In addition to creating secure runtime environments for microservices discussed above, engineers of microservices require a continuous insight into the behavior and health status of each component of the microservices composition. The next section identifies mechanisms to ensure that engineers are aware of the behavior of the components of a microservices composition at all times.

6.6 Continuously monitor components of the microservices composition

In order to troubleshoot security related challenges at runtime, engineers of microservices always require insight into the behavior of each component of the microservices composition. An attack on a component may result in a component responding slowly to requests or becoming unavailable. In a microservices composition, distributed tracing of communication between components and access to each component's log files are vital to understanding attacks. To this end, it is essential to identify the features of each component at design time that are necessary and sufficient to describe and understand the component's runtime security behavior. These features can then be used to determine how any changes in a property will affect the overall status and health of the application. Monitoring metrics can then be defined using the essential features of the components. In a microservices architecture:

- Each component of the microservices architecture, for example, the microservices, service registry and message broker should have its own set of metrics. The metrics define the availability of each component, determine acceptable responses time of each component, identify the origin of each request sent to a component and define how to log errors and exceptions in the application.
- The infrastructure that host the components of a microservices composition should have defined metrics. The infrastructure metrics are those that pertain to the status of the infrastructure and the servers on which the microservice is running. Monitoring docker containers, virtual machines, and networks give an insight into each how each component is using, for example, the CPU, memory, and connections to resources like the database.

The following questions in Table 6.8 are formulated to guide the review.

Table 6.8. Formulated research questions on monitoring mechanisms

Research questions	**Motivation**
REVQ9. What is required to monitor distributed microservices effectively?	To understand the requirements for microservice monitoring
REVQ10. Can available tools assist gain better visibility of microservices and the runtime environment to ensure continuous security?	To gain an understanding of what mechanisms are available to monitor microservices.

REVQ9 is addressed by eliciting the various characteristics that are required to monitor microservices architecture components effectively, described next.

6.6.1 Requirements for security monitoring

The distributed nature of components in a microservices architecture requires a tool that is:

- *Customizable* (Fatema et al., 2014, Sun, Nanda & Jaeger, 2015). In a microservices architecture, different components require different security monitoring metrics. Software engineers of microservices-based applications need tools that allow them to customize the tools to gather different security metrics for each component.
- *Complete.* Continuous security of microservices-based applications such as PickMeUp requires that software engineers have visibility to both the runtime infrastructure, the microservices, service registry, the API gateway, the message broker and the infrastructure. A suitable tool is needed to provide a comprehensive view of the application at runtime.
- *Scalable* (Gogouvitis et al., 2012, Aceto et al., 2013). The number of composed microservices in the application may increase as new business functionality is automated, and therefore a monitoring tool needs to scale as new microservices are added.
- *Portability (*Aceto et al., 2013, Fatema et al., 2014*).* Microservices are portable artefacts deployable on different platforms. Therefore, the ability to use a monitoring tool on a different platform is indispensable.

The next section identifies existing tools that can assist gain better visibility of microservices at runtime to answer REVQ10 in Table 6.8.

6.6.2 Review of existing monitoring tools

The general observation is that the monitoring tools can broadly be classified as follows:

- *Proprietary tools.* Proprietary tools belong to Infrastructure provider or third-party organizations.
- *Free or open-source tools.* Free or open-source tools are freely available monitoring tools.

Table 6.9 provides a summary of a few most common monitoring tools (Fatema et al., 2014). The tools are CloudWatch, CloudMonix, Dynatrace, Zabbix, Prometheus, and AppDynamincs. The summary considers the requirements for monitoring discussed above. Characteristic discussed in Section 6.6.1 are used to review the tools.

Table 6.9. Review of most common monitoring tools

	CloudWatch	**CloudMonix**	**Dynatrace**	**Zabbix**	**Prometheus**	**AppDynamics (Microservices IQ)**
Description	Amazon CloudWatch is a web service that provides real-time monitoring of resources (Amazon, 2017).	CloudMix is an Inbuilt monitoring service for Azure resources (Microsoft, 2014)	Dynatrace monitors the availability and performance of applications and the impact on user experience	Zabbix is an open source monitoring tool for networks, operating systems, and applications (Vacche, 2015)	Prometheus is an open-source monitoring and alerting tool	AppDynamics is an application performance management (APM) tool for monitoring and management of management of software performance (AppDynamics, 2018).
Licensing	Proprietary	Proprietary	Proprietary	Open source	Open source	Proprietary
Types of monitoring	Can be used for performance monitoring, Availability monitoring. Azure log analytics monitors containers (Amazon, 2017).	Can be used for performance monitoring, Availability monitoring. Azure log analytics monitors containers (Microsoft, 2014),	Availability monitoring, Performance monitoring, Container or Virtual machine monitoring.	Availability monitoring, Performance monitoring, Container or Virtual machine monitoring (Vacche, 2015) .	Availability monitoring, Performance monitoring, Container or Virtual machine monitoring.	Availability monitoring, Performance monitoring, Container or Virtual machine monitoring (AppDynamics, 2018).

	CloudWatch	**CloudMonix**	**Dynatrace**	**Zabbix**	**Prometheus**	**AppDynamics (Microservices IQ)**
Customizability	Custom metrics can be sent to CloudWatch using an API (Amazon, 2017).	Supports native integrations to popular third-party platforms like AppDynamics, Nagios, Dynatrace to process information (Microsoft ,2014).	REST API provides means to integrate Dynatrace into continuous deployment pipeline.	Zabbix support custom metrics (Vacche, 2015).	Can be customized by writing new exporter or custom collector.	Alert based on custom validation of HTTP response.
Complete	Can be used to monitor both application and containers. Docker container logs can be retrieved from CloudWatch (Amazon, 2017).	CloudMonix can monitor the containers or virtual machines and the microservices (Microsoft, 2014).	Can monitor microservices and Docker monitoring integrates seamlessly, with no configuration.	Zabbix can monitor application and containers.	Zabbix can monitor application and containers.	AppDynamics monitors both microservices and the runtime environment (AppDynamics 2018).
Scalable	When application auto-scale programmatically, events sent to Amazon CloudWatch (Amazon, 2017).	Provides auto-scaling capabilities for Azure VM (Microsoft, 2014).	Auto-discovers and monitors containers without touching images.	Supports auto-discovery of servers and network devices.	Zabbix may require new installation to handle the new load.	AppDynamics automatically discovers new microservices endpoints.
Portability	Amazon CloudWatch works on both Windows and Linux platforms (Amazon, 2017).	Azure supports a selection of operating systems, programming languages, frameworks, tools, databases, and devices. Runs Linux and Docker containers (Microsoft, 2014)	Supports both Linux and Windows platform.	An agent can be installed on UNIX and Windows hosts. Zabbix also support agent-less monitoring using other protocols like TCP (Vacche, 2015).	Supports both Linux and Windows platform	AppDynamics agents are available for many programming languages, frameworks. An agent can run Linux and Docker containers (AppDynamics, 2018).

6.6.3 General observation and discussion

A review of currently available monitoring tools in Table 6.8, both open source and proprietary, shows that these tools are adequate to monitor microservices, containers and virtual machines. Most of these tools are production-ready. The monitoring tools provide proper logging of all relevant and essential information required by engineers to understand the state of the microservice at any time. The tools discussed in Table 6.8 provide well-designed dashboards that reflect the health of the microservices, organized in a manner that is understandable. The tools provide alerts for critical metrics, which enables engineers to mitigate and resolve problems. With useful logging, dashboards, alerting, the microservice's availability can be protected, failures and errors detected and reduced. With many tools to choose from, the following security decisions should be considered:

- The limitation of agent-based tools such as AppDynamics, Dynatrace, and Zabbix is that they often rely on installing an agent on the container or virtual machine. When the container or virtual machine is compromised, the results collected from monitoring may not be trustworthy (Sun, Nanda & Jaeger, 2015). Monitoring for security should be built based on the assumption that the runtime environment hosting microservices cannot be trusted because it is possible for an attacker to gain control over it.
- Tools used for monitoring should not introduce any vulnerabilities. Monitoring capabilities such as user-based access control, secure notification and storage are essential.

In addition to the security considerations above, the following limitations should be considered when selecting a tool:

- Proprietary tools may result in vendor lock-in as they are part of the technology stack of a vendor. For example, CloudWatch is a proprietary monitoring tool for Amazon cloud services while Azure watch is a is a proprietary tool for monitoring Microsoft cloud services.
- Proprietary tools require license fees.
- Open-source tools can be hard to use. The user of the tools should make sure the tools are set up to be resilient and scalable.

In addition to monitoring microservices, the final security development requirement is to ensure that microservices respond securely to changes in their runtime environment. The next section reviews methods that enable microservices to react safely to changes in their environment at runtime.

6.7 Securely respond to attacks using adaptation mechanisms

Microservices-based applications are distributed systems implemented using a collection of components that independently evolve and react to their environments and other external factors. The distributed nature of components creates many potential points of failures. Failures could be a result of a security attack or a component becoming unavailable due to technical reasons. The premise of developing secure microservices-based applications is the ability to withstand failures and expected attacks. The applications should be built with failure in mind to ensure that the applications can respond securely to both internal as well as external changes that can affect the security posture. Reaction to changes should not involve manual intervention. The reaction to changes without manual intervention is called self-adaption. Secure self-adaptation in microservices is the achievement of a safe, stable and desirable configuration in the presence of malicious attacks. The below question is used to survey the literature on microservices secure self-adaptation.

Table 6.10. Review of most common monitoring tools

Research question	Motivation
REVQ11. How can microservice be designed to withstand a failure in the event of a security attack?	To understand the requirements and available methods to achieve secure self-adaptation of microservices.

The next section identifies elements for self-adaptation.

6.7.1 Requirement for secure self-adaptation

Designing microservices-based application for failure in the event of a security attack require the application to possess the following capabilities:

- *Self-protection:* The ability of microservices to secure themselves against potential attacks (Behringer et al., 2015).
- *Self-configuration:* Requires the microservices to securely reconfigure themselves, based on self-knowledge, discovery, and intent (Behringer et al., 2015).

The following approaches can achieve self-protection and self-configuring of the microservices composition:

- *Ensuring availability of components* (Dragoni et al., 2017) -given that a microservice, the service registry, the gateway and message broker can be unavailable at runtime as a result of an attack, deploying replicas of the component can assist in ensuring availability. When one instance is unavailability, the replica can service requests.
- *Self-healing capabilities* (Toffeti, 2015) - self-healing is the ability of a component in the microservices architecture to restart itself if the component is not in health status. A microservices-based application can take advantage of tools like Kubernetes (Brewer, 2015). These tools provide functionality to cluster containers.
- *Isolating failures* (Fontesi & Weber, 2016) - microservices should be equipped with the ability to fail fast or provide a fallback if the microservices is unavailable or the response time is slow. Example of implementation is to use the circuit breaker pattern (Fontesi & Weber, 2016). The Netflix Hystrix (Fontesi & Weber, 2016) is an example of an implementation that uses the circuit breaker pattern.

6.7.2 General observations and discussion

A review of the literature on self-adapting microservice shows that security is briefly addressed (Sun, Nanda & Jaeger, 2015). The technology to enable self-healing of microservices is new and not yet mature. Tools have not yet matured to be used to address the security of microservices. Tools such as Kubernetes are still on the Alpha-support level for Windows platforms (Kubernetes, 2018). The Netflix Hystrix library used to isolate failures is only available to microservices built using Java. Only the option to ensure component availability using clustering and load balancing is established in the software development community. There is reliance on cloud vendor to achieve scalability (Sun, Nanda & Jaeger, 2015).

The section above has discussed the various security activities that are required to create a secure microservices-based application. The next section provides a summary of the findings.

6.8 Summary

The microservices architecture decomposes applications into a myriad of small, distributed and conversational microservices that cannot be designed to trust each other entirely. The compromise of one component may bring down the entire application. Given this background, this chapter reviewed six security activities that are required to ensure that microservices compositions are designed and developed for security. Table 6.11 below summaries the review and identify the research gap.

Table 6.11. Summary of the review of security activities

Security activity	**Research questions**	**Findings**	**Summary**	**Potential research**
1. Define a security policy of microservices composition	**REVQ1.** What is a good microservices security policy design?	1. Support for distributed and loosely coupled microservices interactions 2. Support for hierarchical security policy domains	1. There is currently no policy languages and a standard to specify a security policy for RESTful web services 2. engineers need to define a microservices composition security policies manually	Creating a hierarchical security policy model for a microservices composition
	REVQ2. What categories of security policies apply to a microservices architecture?	1. Data protection policy 2. Access control policy 3. Microservice technology-specific policy 4. Network security policy 5. Microservice composition security policies 5.Virtual Machine and container policy		
2. Adopting secure programming practices	**REVQ3.** Which secure programming practices can assist engineers to avoid known security flaws when developing microservices?	1. Keep the microservices design simple 2. Ensure input validation on the microservices API 3. Give attention to compiler warnings 4. Sanitize data sent to other microservices 5. Adhering to the principle of least privilege 6. Practicing defense in depth 7. Denying access by default. 8. Ensuring secure deployment pipelines	1. Each year, most of the top ten web application vulnerabilities published by OWASP are known vulnerabilities 2. Software engineers and architect are reluctant to adopt secure programming practices 3.A small fraction of software engineers are well trained in secure software development	Formulate practice-oriented methods to integrate safe programming practices into the software development process of microservices

Security activity	**Research questions**	**Findings**	**Summary**	**Potential research**
2. Adopting secure programming practices (continued)	**REVQ4.** Can secure programming practices be enforced without affecting the benefits of continuous delivery and agile methods?	1. Translating secure programming practices into secure coding standards or were possible microservices security requirements 2. Documenting the security requirements into security policies 3. Using the continuous delivery toolchain as a point were violations of secure programming practices, and security requirements are identified and attended to		
3. Security testing	**REVQ5.** What characteristics should a security testing tool posses to effectively integrate into a continuous delivery and agile development environment?	1. Ease to integrate 2. Easy to use 3. Easy results interpretation 4. Extensive language support and portability 5. Extensibility	1. There is a significant dependency on external vendors to perform penetration testing 2. Little attention is paid to perform security testing across the entire life-cycle of the development process in agile teams	Create a framework to ensure systematic integration of security testing into all phases of the software development life-cycle
	REVQ6. Can existing security testing tools be used to automate security testing in continuous delivery and agile practices?	SonarQube (Campbell & Papapetrou 2013).Gauntlt (Koc et al. 2017)FindSecurityBugs (Arteau 2016)		
4. Creating secure runtime environment configurations	**REVQ7.** What capabilities should a platform or tool possess for secure configurations?	1. Support for diverse security configuration use cases 2. Allow authoring and version control of security configurations 3. Validation of configurations 4. Easy management dependencies between configuration files 5. Low learning curve	1. Minimal effort has been made to use the various management tools to ensure secure configurations 2. There is a lack of documented precedent when it comes to using configuration management tools for an audit against security standards	Leverage existing configuration management tools to create programmable, automated and template-based security configurations

Security activity	**Research questions**	**Findings**	**Summary**	**Potential research**
4. Creating secure runtime environment configurations (continued)	**REVQ8.** Can the widely used configuration management tools be leveraged to create secure microservices runtime environment?	Tools like Chef, puppet, Ansible can be used to configure secure runtime environment for microservices		
5. Monitoring mechanisms	**REVQ9.** What is required to monitor distributed microservices effectively?	1. Customizability 2. Complete 3. scalability 4. Portability	Both open source and proprietary tools are adequate to monitor microservices and the containers. Most of these tools are production-ready	
	REVQ10. What available tools can assist gain better visibility of microservices at runtime?	Example include CloudWatch, CloudMonix, Dynatrace, Zabbix, Prometheus, AppDynamics (Microservices IQ)		
6. Secure microservices adaptation	**REVQ11.** How can microservices be designed to withstand a failure in the event of a security attack?	1. Ensuring availability of components using clustering technologies 2. Using container orchestration tools for secure self-healing 3. Isolating failures using the circuit breaker pattern	The technology to enable self-healing is not yet mature and lacks cross-platform support	

Table 6.11 above shows that although little attention has been paid to develop secure microservices-based applications, many tools and methods are available that can be used to ensure end-to-end security. There is therefore a need to create a framework to ensure systematic integration of various tools, techniques and methods into all phases of the software development life-cycle to ensure security.

6.9 Conclusion

The review in this chapter has shown that the development of a secure microservices composition that can continue to function securely under malicious attacks is a complex exercise. Despite the challenge, the goal of creating secure microservices compositions can be achieved when a set of secure development activities that focus on the security aspects of a composition are integrated early into the development process. First, there is a need to ensure precise documentation of security requirements using various types of security policies. The security policy comprehensively captures all the protection measures of various components of the compositions. Then, during the design and development of each microservices many security vulnerabilities resulting from poorly constructed microservices should be avoided by adopting secure programming practices. The integration of safe programming practices should be done in a manner that is friendly to software developers with little security expertise, and also in a way that does not affect the productivity of software developers.

Microservices compositions applications can also be inherently secure when engineers exploit new opportunities that security testing tools bring into the microservices development environment. Conducting security testing early in the development process allow security vulnerabilities to be identified. Identified security vulnerabilities can then be addressed early in the development process. Code-driven, configuration management tools, should also be adapted to provide standardized, secure configurations of microservices and their runtime environments using commonly tested templates. Microservices compositions can be made more secure by using mechanisms to detect any anomalies at runtime which can compromise security.

The next chapter discusses the development of a framework to ensure microservices that provide end-to-end security. The framework incorporates practice-oriented methods to integrate safe programming practices into the software development process of microservices and ensure systematic integration of security testing into all phases of the software development life-cycle. The framework leverages existing configuration management tools to create programmable, automated and template-based security configurations.

PART III

Chapter 7

SAFEMicroservices A Development Framework for Secure Microservices Compositions

7.0 Introduction

Secure software development approaches have been created in the past to assist software engineers. However, many organizations find the cost associated with using existing secure software development approaches or methodologies prohibitive (Geer, 2010, Viega, 2011). Secondly, existing secure software development approaches are designed for sequential software development methodologies (Baca & Carlsson, 2011, Jøsang, Ødegaard & Oftedal, 2015). In this regard, it is difficult for software engineers to apply these secure software development approaches in iterative software development methodologies such as the Agile methodology. Furthermore, these existing secure software development approaches were designed for siloed software development environments were roles of software engineers and infrastructure engineers are not tightly coupled (De Win et al., 2009). On the contrary, DevOps, a new trend in software development, advocate for a multidisciplinary team working together in a fast-paced manner. The existing secure development approaches, therefore, fall short when developing secure microservices compositions in fast-paced environments. Accordingly, this chapter proposes a practice-oriented framework to assist software engineers in fast-paced teams in developing secure microservices compositions from the ground up. The framework is called *SAFEMicroservices.*

The fundamental premise of SAFEMicroservices is that security should inherently be built in a microservices composition by using a robust architecture design (Feng et al., 2016, Santos, Tarrit & Mirakhorli, 2017). SAFEMicroservices is based on the argument that the architecture of microservices should incorporate design decisions that promote security. This argument is based on a concept called *secure-by-design* discussed in Chapter 5. Furthermore, SAFEMicroservices is based on the view that when software engineers adopt appropriate security-focused tools and techniques as part of their daily software development tasks, a coordinated security strategy can be created that cultivates a security-conscious culture among software engineers. A security-conscious culture becomes an essential quality that enables software engineers to build secure and resilient microservices compositions (AlHogail, 2015). SAFEMicroservices is generic in approach to security but flexible enough in its application so that it accommodates variations in the implementation using different technologies, software development methodologies, and also considers the risk profile of the intended microservices composition.

The organization of this chapter follows the following sequence: Section 7.1 introduce the aims of the SAFEMicroservices. Section 7.2 discusses SAFEMicroservices in detail. Section 7.3 summaries the benefits of the SAFEMicroservices approach. A conclusion then follows in section 7.4

7.1 Aims of the SAFEMicroservices framework

In addition to addressing the lack of a light-weight secure software development methodology that can be used in a fast-paced development team, the SAFEMicroservices framework aims to contribute the following to the development of secure microservices.

a) Comprehensive analysis of security threats and vulnerabilities associated with the microservices architectural style

Although studies have been undertaken on microservices security, these have been piecemeal approaches (Sun, Nanda & Jaeger, 2015, Yarygina and Bagge, 2018). Such studies are largely not based on design-level security considerations which requires a deep understanding of the application's architecture to uncover design flaws. Design flaws are often one of the underlying causes of the difficulties faced by software engineers when implementing software patches in the event of security breaches (Feng et al., 2016). With this in mind, the foundation of SAFEMicroservices is a comprehensive analysis of the security threats and vulnerabilities rooted in the microservices architectural style.

b) Comprehensive set of coding guidelines focused on a secure architecture of microservices

Although various guidelines are available in the literature to assist software engineers to write secure code, such documentation tends to be platform or language-specific or generic. Example of such guidelines includes coding guidelines for the Java platform (Long et al., 2011), the Microsoft .NET platform (Howard & LeBlanc, 2003) or very general guidelines such as the one provided by the Open Web Application Security Project (OWASP) (OWASP, 2011). Such platform or language-specific guidelines do not provide software engineers with sufficient guidance to avoid subtle architecture-level security threats and vulnerabilities. In this regard, SAFEMicroservices presents a microservices architecture-driven identification and classification of secure coding guidelines to augment existing secure coding guideline documentation.

c) Assist software engineers to incorporate security in the microservices development lifecycle

Although incorporating security requirements in early phases of microservices software development is considered the most cost-effective way to develop secure applications (Souag et al., 2016), software engineers tend to prioritize functional requirements at the expense of security (Oyetoyan et al., 2016). The reason is that it is difficult for software engineers to justify the incorporation of security-activities from a customer or end-user perspective during the planning phase (Oyetoyan et al., 2016). SAFEMicroservices incorporate mechanisms to represent security requirements in a manner that show tangible business impact and promote security awareness in the organization.

The SAFEMicroservices framework intends to achieve its aims by ensuring that the following five objectives are met:

- Software engineers are encouraged to become more focused on identifying and mitigating security flaws and weaknesses in the design of microservices compositions as studies have shown that many applications are breached due to weaknesses in their design (IEEE Center for Secure Design, 2015).
- Software engineers are offered guidance on how to incorporate security-oriented activities, tools, and techniques in their daily software development tasks (Cruzes et al., 2017).
- Security awareness is promoted among software engineers and other stakeholders in the organization in a manner that makes software security to be accommodated in any technology migration plan.
- Software engineers get timely feedback on security vulnerabilities on the microservices source code during software coding, as soon as software changes are committed into the version control system, and also when software is deployed.

- Reusable microservices security artefacts are created early in the software development lifecycle to make replication of microservices security successes easier in subsequent microservices-based projects.

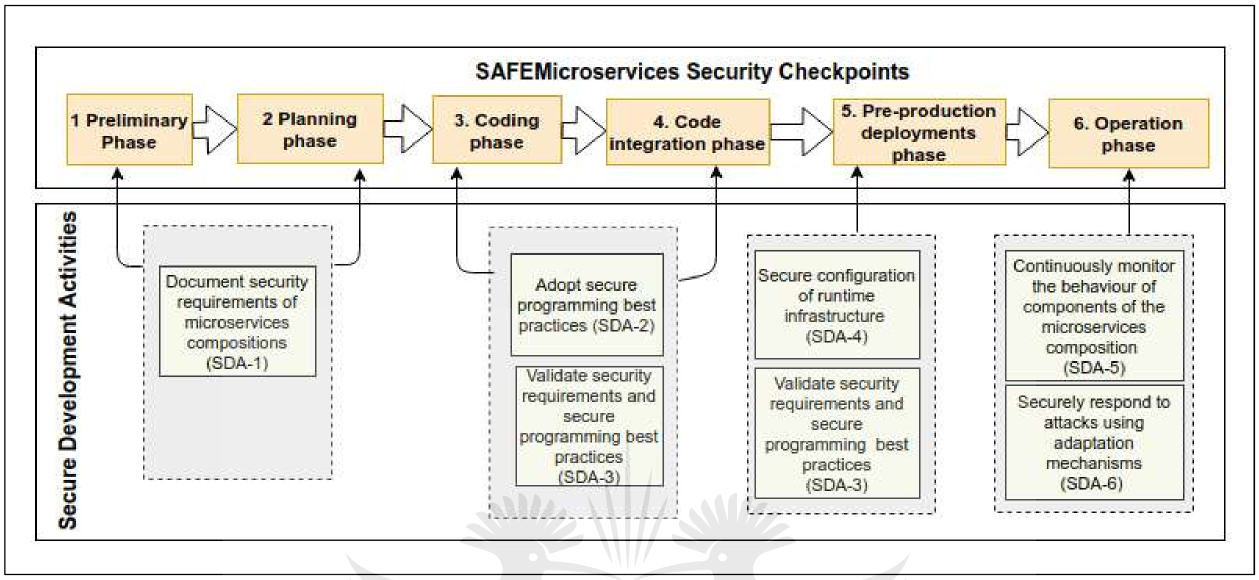
The next section discusses the SAFEMicroservices framework in detail.

7.2 SAFEMicroservices framework

To understand how to develop microservices compositions that are inherently safe, Chapter 5 identified six general secure development activities that provide the necessary foundation for SAFEMicroservices. SAFEMicroservices aims to assist software engineers to integrate the six secure development activities into their software development process. To this end, SAFEMicroservices identifies six critical phases in the software development process that are common to sequential, incremental and iterative methodologies, thereby ensuring that new trends in software development are applied. These six phases are the points where the six development activities are integrated into the development process. These phases are referred to as *security checkpoints* in SAFEMicroservices. The SAFEMicroservices framework security checkpoints are briefly defined below:

- *Preliminary phase* – the preliminary phase is a phase common to most software projects where customer requirements and the system's use cases are documented. The phase is also used to create the development infrastructure.
- *Planning phase* – planning phase is the period when the software development team identifies the scope of what needs to be delivered by defining software requirements.
- *Coding phase* – the coding phase is the period when software engineers develop software using an integrated development environment (IDE).
- *Code integration phase* – the code integration phase is when all software changes written by software engineers are combined and built to create a deployable artefact.
- *Pre-production deployment phase* – the pre-production deployment phase is when software artefacts are deployed to pre-production environments for quality assurance.
- *Operational phase* – operational phase is when the software artefact is deployed to a production environment and released to end-users.

Figure 7.1 below shows where SAFEMicroservices *security checkpoints* are defined for development phases to perform the six secure microservices development activities identified in Chapter 5, that are denoted by SDA1 – SDA6 (Secure Development Activity).



SAFEMicroservices Security Checkpoints
1 Preliminary Phase
2 Planning phase
3. Coding phase
4. Code integration phase
5. Pre-production deployments phase
6. Operation phase
Secure Development Activities
Document security requirements of microservices compositions (SDA-1)
Adopt secure programming best practices (SDA-2)
Validate security requirements and secure programming best practices (SDA-3)
Secure configuration of runtime infrastructure (SDA-4)
Validate security requirements and secure programming best practices (SDA-3)
Continuously monitor the behaviour of components of the microservices composition (SDA-5)
Securely respond to attacks using adaptation mechanisms (SDA-6)

Figure 7.1: Secure development activities in SAFEMicroservices

As indicated above, SAFEMicroservices aims to be generic, to be able to support any type of developmental methodology such as sequential, incremental and iterative methodologies. In this regard, Figure 7.2 shows how the SAFEMicroservices security checkpoints can be used in both an iterative and sequential software development methodologies. On the left, the SAFEMicroservices framework is applied to a sequential development methodology. On the right, an iterative methodology such as the Agile methodology is shown where the preliminary phase is first completed, and then the planning, coding, code integration, and pre-deployment.

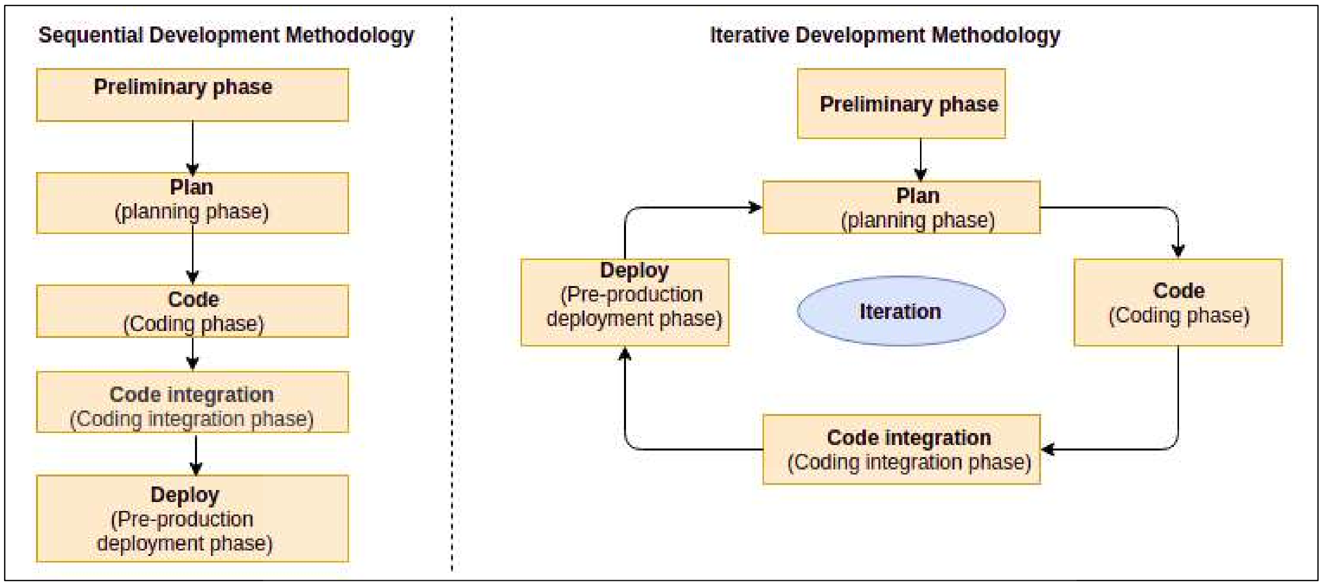

Sequential Development Methodology
Preliminary phase
Plan (planning phase)
Code (Coding phase)
Code integration (Coding integration phase)
Deploy (Pre-production deployment phase)
Iterative Development Methodology
Preliminary phase
Plan (planning phase)
Deploy (Pre-production deployment phase)
Iteration
Code (Coding phase)
Code integration (Coding integration phase)

Figure 7.2. security control gates in an iterative and sequential methodology

Next, Figure 7.3 below presents a detailed view of the complete SAFEMicroservices framework. The security checkpoints and the vital secondary activities that software engineers should perform to support the six secure development activities is given. In addition, Figure 7.3 shows both the input required for each secondary activity and the reusable security artefacts produced by the activities.

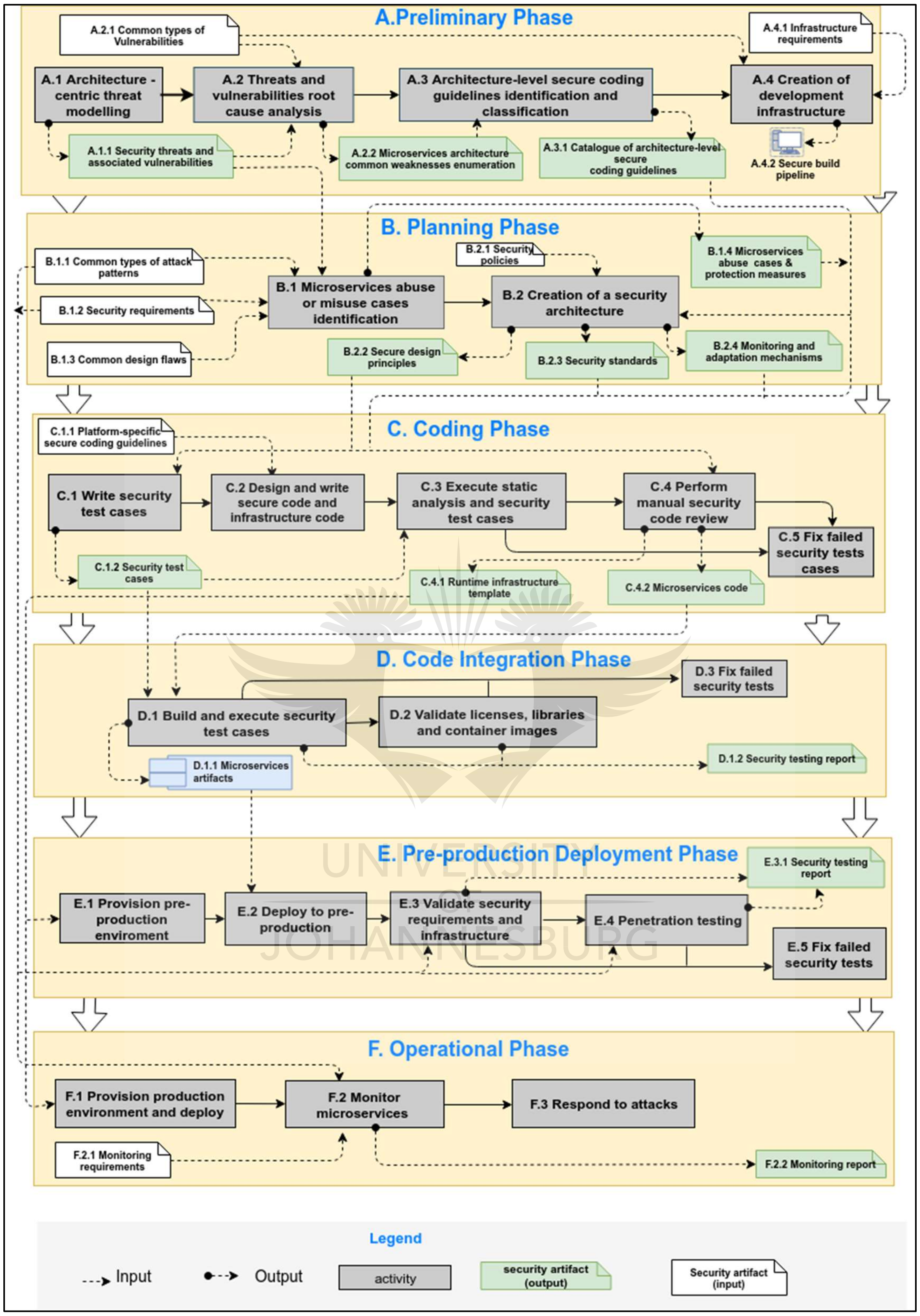

A.Preliminary Phase
A.2.1 Common types of Vulnerabilities
A.4.1 Infrastructure requirements
A.1 Architecture - centric threat modelling
A.2 Threats and vulnerabilities root cause analysis
A.3 Architecture-level secure coding guidelines identification and classification
A.4 Creation of development infrastructure
A.1.1 Security threats and associated vulnerabilities
A.2.2 Microservices architecture common weaknesses enumeration
A.3.1 Catalogue of architecture-level secure coding guidelines
A.4.2 Secure build pipeline
B. Planning Phase
B.2.1 Security policies
B.1.1 Common types of attack patterns
B.1.4 Microservices abuse cases & protection measures
B.1.2 Security requirements
B.1 Microservices abuse or misuse cases identification
B.2 Creation of a security architecture
B.1.3 Common design flaws
B.2.2 Secure design principles
B.2.3 Security standards
B.2.4 Monitoring and adaptation mechanisms
C. Coding Phase
C.1.1 Platform-specific secure coding guidelines
C.1 Write security test cases
C.2 Design and write secure code and infrastructure code
C.3 Execute static analysis and security test cases
C.4 Perform manual security code review
C.5 Fix failed security tests cases
C.1.2 Security test cases
C.4.1 Runtime infrastructure template
C.4.2 Microservices code
D. Code Integration Phase
D.3 Fix failed security tests
D.1 Build and execute security test cases
D.2 Validate licenses, libraries and container images
D.1.1 Microservices artifacts
D.1.2 Security testing report
E. Pre-production Deployment Phase
E.3.1 Security testing report
E.1 Provision pre-production enviroment
E.2 Deploy to pre-production
E.3 Validate security requirements and infrastructure
E.4 Penetration testing
E.5 Fix failed security tests
F. Operational Phase
F.1 Provision production environment and deploy
F.2 Monitor microservices
F.3 Respond to attacks
F.2.1 Monitoring requirements
F.2.2 Monitoring report
Legend
Input
Output
activity
security artifact (output)
Security artifact (input)

Figure 7.3. SAFEMicroservices

The SAFEMicroservices framework consists of six phases that need to be integrated with the phases of a relevant software development framework or methodology. There is a total of twenty-one security activities that need to be performed, seven security artefacts required as input to the activities, and twelve resultant security artefacts as output. It is vital to point out at this stage that Figure 7.3 above is not meant to depict SAFEMicroservices as a sequential process using phases but merely to group and show the various activities that should be performed at a phase in any software development methodology.

Next, each SAFEMicroservices phase is discussed in detail using Figure 7.3 above.

7.2.1 Preliminary phase

SAFEMicroservices preliminary phase aims to perform a risk assessment of the microservices composition and also pro-actively creates the necessary development infrastructure that software engineers can use to write and test microservices. Figure 7.4 give the security-focused activities and security artefacts of the SAFEMicroservices preliminary phase, as defined by the preliminary phase shown in Figure 7.3.

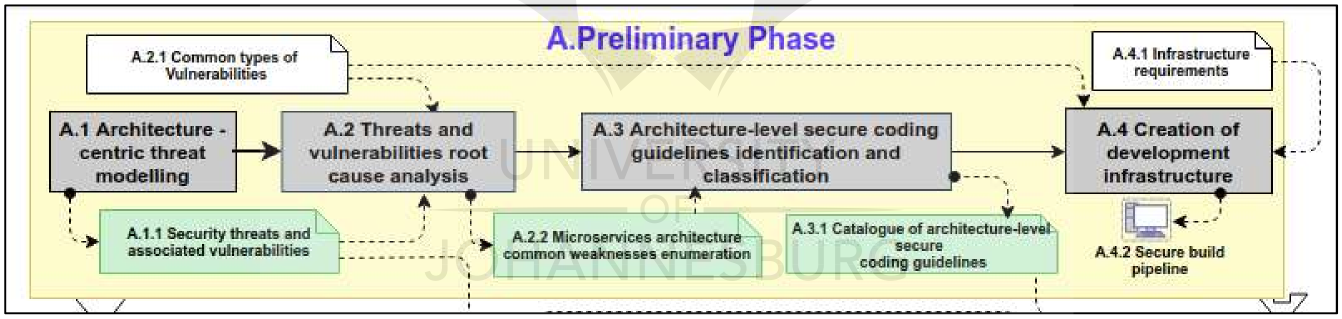

A.Preliminary Phase
A.2.1 Common types of Vulnerabilities
A.4.1 Infrastructure requirements
A.1 Architecture - centric threat modelling
A.2 Threats and vulnerabilities root cause analysis
A.3 Architecture-level secure coding guidelines identification and classification
A.4 Creation of development infrastructure
A.1.1 Security threats and associated vulnerabilities
A.2.2 Microservices architecture common weaknesses enumeration
A.3.1 Catalogue of architecture-level secure coding guidelines
A.4.2 Secure build pipeline

Figure 7.4. SAFEMicroservices preliminary phase activities

The security focused-activities of the preliminary phase shown in Figure 7.4 are secondary activities that aim to address the need to document the security requirement of a microservices composition (SDA-1). These four secondary activities of the preliminary phase are discussed next.

(a) A.1 Architecture-centric threat modeling

An architecture-centric approach is used in SAFEMicroservices to unravel and understand security threats and vulnerabilities of the microservices architectural style. The approach assists to identify security weaknesses in individual microservices and those that can occur when the various component of a microservices composition interact. Architecture-centric threat modeling performed in Chapter 5 identified five microservices composition security threats namely:

1. Insecure application programming interfaces,
2. Unauthorized access,
3. Insecure service discovery,
4. Insecure message broker and
5. Insecure runtime infrastructure.

The risk assessment performed in Chapter 5 identified vulnerabilities associated with each threat by reviewing common vulnerabilities and exposures (CVE), a dictionary of common names for publicly known security vulnerabilities. The result is presented as an artefact that is adapted as new vulnerabilities are identified.

Produced security artefacts:

A.1.1 Security threats and associated vulnerabilities.

The next step in SAFEMicroservices is to perform a detail analysis of the security threats and vulnerabilities.

(b) A.2 Threats and vulnerabilities root cause analysis

The aim of the threat and vulnerabilities root cause analysis in SAFEMicroservices is to first understand the design decisions that are the root cause of microservices vulnerabilities. This understanding can assist software engineers to identify appropriate architecture-level guidelines that can be used to avoid both insecure designs and poor implementation of the microservices architecture. Secondly, the analysis seeks to assist software engineers to gain a deeper understanding of attack strategies that a malicious user can employ to exploit the weakness. Such knowledge enables engineers to anticipate threats and devise protection measures accordingly during the implementation of microservices. The security artefact employed are shown next.

Required Security artefacts:

A.1.1 Security threats and associated vulnerabilities

A.2.1 Common types of vulnerabilities

The artefact that is produced by threat and vulnerabilities root cause analysis is shown next.

Produced Security artefacts:

A.2.2 Microservices architecture common weaknesses enumeration (MACWE)

The analysis of threats and vulnerabilities uses *security arterfact A.1.1*, identified in the previous step. The analysis augments this list to be more comprehensive by including common types of vulnerabilities as denoted by *security artefact A.2.1*. The list of common types of vulnerabilities is retrieved from a community-developed dictionary of software weakness types (CWE) database (Mitre 2018) that provide specific and succinct definitions of common weaknesses. In the CWE dictionary, a vulnerability is referred to as a *software weakness*. The CWE database categorises software weakness according to architecture concepts. Each weakness is prefixed with CWE and is assigned a number. An architectural concept is a classification criterion that groups vulnerabilities according to common design decisions in the system architecture (Mitre 2018). Table 7.1 shows a list of architecture concepts from CWE, as adapted for the context of microservices.

Table 7.1. Architecture concepts

Architecture concepts adapted from CWE list
• *Validate input* – A category of vulnerabilities related to how the microservices composition handles input validations • *Authorize actors* – A category of vulnerabilities related to how a microservices composition handles authorization • *Limit exposure* – A category of vulnerabilities related to the microservices entry points • *Limit access* – A category of vulnerabilities related to the way microservices limit access to infrastructure components • *Encrypt data* -A category of vulnerabilities related to how the microservices composition ensures confidentiality • *Audit* – A category of vulnerability related to the way the microservices composition handle logging of user activities • *Authenticate actors* – A category of weakness related to how a composition handles authentication • *Identify actors* – A category of weakness related to how the composition identify a user • *Verify message integrity* – A category of vulnerabilities related to the way the microservices ensure data integrity

The architecture concepts in the CWE taxonomy that apply to the microservices composition security threats as identified in the architecture-centric threat modelling is now identified, as shown in Table 7.2. This evaluation gives an understanding of common design decisions in the microservices architecture and the related security threats. The microservices architectural components are also shown.

Table 7.2. CWE architecture concepts applicable to microservices security threats

Security threats	**Applicable architecture concepts**	**Microservices components**
Insecure application programming interfaces	• Validate input • Authorize actors • Authenticate actors • Limit exposure • Encrypt data • Verify message integrity • Audit	• API gateway • All microservices
Unauthorized access	• Authorise actors	• All components
Insecure microservice discovery	• Limit exposure • Encrypt data • Verify message integrity • Validate input • Authorize actors • Authenticate actors • Audit	• Service registry, Microservices
Insecure runtime infrastructure	• Limit access • Limit exposure • Authorise actors	• Runtime infrastructure

Insecure message broker	• Limit exposure • Encrypt data • Limit access • Verify message integrity	• Message broker

Next, the vulnerabilities classified under each CWE architectural concept is retrieved. These vulnerabilities are analysed to gain a deeper understanding of the nature of the weakness. Any security vulnerabilities classified under the architecture concept but not relevant to the microservices architecture are eliminated. The analysis allows the CWE dictionary to be extended into a microservices architecture taxonomy of common weakness types. As an example, Table 7.3 below shows the vulnerabilities related to the *validate input* architectural concept. Also indicated are elicited vulnerabilities that are applicable to security threats. A tick (√) indicates the vulnerability is applicable, and an *x* means the vulnerability does not apply. A comprehensive table is provided in Appendix A – *security artefact A.2.2.*

Table 7.3. Microservices architecture common weakness enumeration

		Services security threats				
Architecture category	**Common vulnerabilities**	**Insecure API**	**Unauthorized access**	**Insecure microservices discovery**	**Insecure runtime infrastructure**	**Insecure message broker**
Validate Input	Improper input validation (CWE-20)	√	x	√	x	x
	Improper neutralization of request data (CWE-138,150, 643, 74, 76, 77, 78, 943, 95, 96, 93)	√	x	√	x	√
	Acceptance of extraneous untrusted data with trusted data	√	x	√	x	√
	Cross-site request forgery (CSRF) (CWE-352)	√	x	x	x	√
	Deserialization of untrusted data (CWE-502)	√	x	√	x	√
	Failure to sanitize special elements in request data (CWE-75,)	√	x	√	x	x
	Improper filtering of request data (CWE-790,791,792, 795,796,797)	√	x	√	x	x
	Argument injection mechanisms (CWE-88)	√	x	√	x	x
	XML injection (CWE-91)	√	x	√	x	√

The microservices architecture common weaknesses enumeration (MACWE), numbered as A.2.2, is an important artefact in SAFEMicroservices. The security artefact aims to assist software engineers to understand the architecture design decisions that apply to each microservices security threat and the associated software weaknesses. Software engineers can use this artefact as a cheat sheet of software weaknesses to avoid when developing a microservices architecture component during the coding phase discussed later below. The elicitation of architecture-level secure coding guidelines and classification in SAFEMicroservices is now discussed below.

(c) A.3 Architecture-level secure coding guidelines identification and classification

The identification of secure coding guidelines makes use of the microservices architecture common weaknesses enumeration security artefact produced by the threats and vulnerabilities root cause analysis. Each of the weakness in the microservices architecture common weaknesses enumeration is thoroughly reviewed to identify design and implementation strategies that software engineers can use to avoid the weaknesses. Table 7.4 shows the structure of the table used to present the guidelines. For example, guidelines for *Improper input validation (CWE-20)* weakness type is shown as an illustration. The architecture-level secure coding guidelines identification and classification activity require the following security artefact below.

Required Security Artefact:

A.2.2 Microservices architecture common weaknesses enumeration (MACWE)

The outcome of the architecture-level secure coding guidelines identification and classification activity is the following artefact.

Produced Security Artefact:

A.3.1 Catalog of architecture-level secure coding guidelines

A comprehensive list of architecture-level secure coding guidelines elicited in this activity is provided in Appendix A – security artefact A.3.1.

Table 7.4. Example of a catalog architecture-level secure coding guidelines

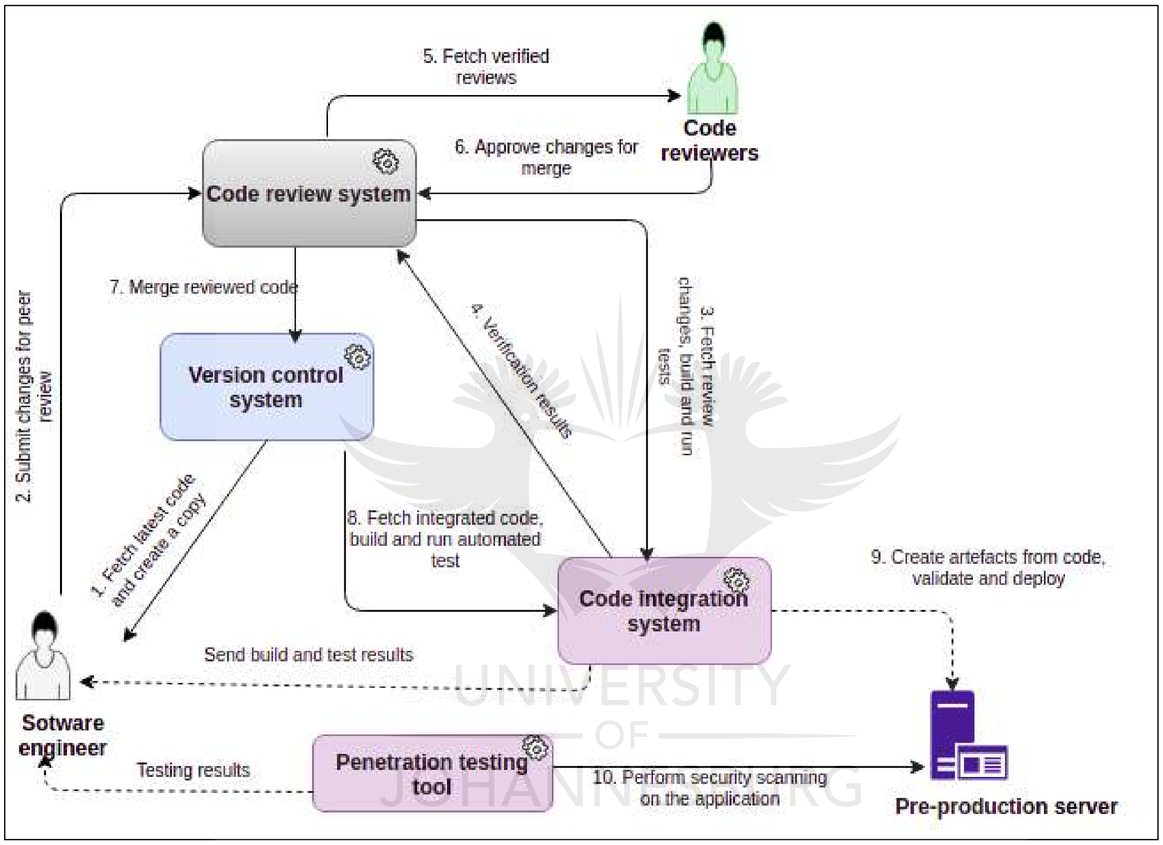
CWE architecture concepts	**Common vulnerabilities**	**Introduction phase**	**Architecture-level secure coding guidelines**
Validate input	Improper Input Validation *(CWE-20)*	Architecture and Design, Implementation	• Validate all inputs - validation should consider relevant properties such as length, input type, and acceptable values • Use and specify an output encoding that is supported by a downstream component that consumes its output • Decoded and canonicalize inputs to the application's current internal representation before validated

The catalog of architecture-level secure coding guidelines produced in the preliminary phase is an essential reusable artefact in SAFEMicroservices. The artefact is intended to assist software engineers to avoid creating vulnerable microservices designs.

(d) A.4 Creation of development infrastructure

The creation of development infrastructure is administrative in nature and ensures that the necessary microservices development infrastructure required by software engineers is in place before the writing of microservices source code commences. Infrastructure requirements are used to guide the creation of the infrastructure in order to ensure that the infrastructure meets both the performance, security and scalability standards and is also not the weakest link that an attacker can use to compromises the microservices artefacts that ultimately get deployed. Figure 7.5 below shows a high-level design of the *SAFEMicroservices* development infrastructure that should be set up.

The infrastructure proposed in Figure 7.5 enables software engineers to acquire (1) microservices source code from a secure repository to make the necessary software changes. Once software changes are completed, the engineer should be able to submit the changes for review (2) to the code review system. The code integration system fetches the changes from the code review system (3), builds the changes, and executes security test cases and submit the verification results to the code review system (4). The reviewer, another software engineer, retrieves the verification results from the code review system (5), and if the feedback of the test execution done by the code integration system is positive, the reviewer can proceed with a manual review of the code changes to check for any security issues on the changes that were not identified by the code integration system. If the changes do not violate any security requirements, secure coding guideline, and secure design principles, the reviewer approves (6) the software changes for integration. The changes are then integrated into the main code base (7), and the code integration system fetches the integrated source code (8), builds the code, executes the security test cases and if all is successful, the integration system build and validate the artefact (9), and then the artefact can be deployed to pre-production environments for quality assurance.



5. Fetch verified reviews
Code reviewers
6. Approve changes for merge
Code review system
2. Submit changes for peer review
7. Merge reviewed code
Version control system
4. Verification results
3. Fetch review changes, build and run tests
1. Fetch latest code and create a copy
8. Fetch integrated code, build and run automated test
9. Create artefacts from code, validate and deploy
Code integration system
Send build and test results
Sotware engineer
Testing results
Penetration testing tool
10. Perform security scanning on the application
Pre-production server

Fig 7.5. SAFEMicroservices development infrastructure high-level design

SAFEMicroservices does not dictate the tools to be used. However, the infrastructure should meet the following basic standards:

- Access to the version control system, code integration system and code review system should be secured through well-defined authentication and authorisation mechanisms. The activities of software engineers on each of the system should be tracked and auditable.
- Software engineers should adhere to a source code branching strategy that adheres to best practices when using the version control. Branching, in revision control is the duplication of

source code so that modifications can happen in parallel along both branches. The originating branch is called the parent branch. Source code should only be merged into the parent branch after a rigorous security testing and peer code review and only source code from the parent branch should be deployed.

(e) Summary of preliminary phase deliverables

The outcomes of the preliminary phase are important re-usable security artefacts that software engineers can use in any microservices-based project. The artefacts are summarized below in Table 7.5.

Table 7.5. Summary of SAFEMicroservices preliminary phase deliverables

SAFEMicroservices activity	Activity deliverable
Architecture-threat modeling	A list of security threats and associated vulnerabilities
Threats and vulnerabilities root cause analysis	A catalog of microservices architecture common weakness classified by threat type, affected security properties or architectural concepts
Architecture-level secure coding guidelines identification and classification	A catalog of architecture-level secure coding guidelines classified by threat type, affected security properties, and architecture concepts
Creation of development infrastructure	Secure build pipeline that software engineers can confidently use to build microservices and perform comprehensive security testing quickly

The activities of the preliminary phase ensure that SAFEMicroservices meet the objectives of making sure that software engineers focus more on identifying and mitigating security flaws and weaknesses in the design of microservices compositions. Besides, the deliverables of the preliminary phase are re-usable artefacts that are essential in the development of microservices.

The next section discusses the SAFEMicroservices planning phase.

7.2.2 Planning phase

The purpose of the SAFEMicroservices planning phase is to construct comprehensive design artefacts that describe how microservices security controls are positioned to maintain confidentiality, integrity, availability, and non-repudiation at all times. The description of security controls concerning the security properties of the microservices composition form the security architecture of the microservices composition. Figure 7.6 below shows the two main security-focused activities of the SAFEMicroservices planning phase.

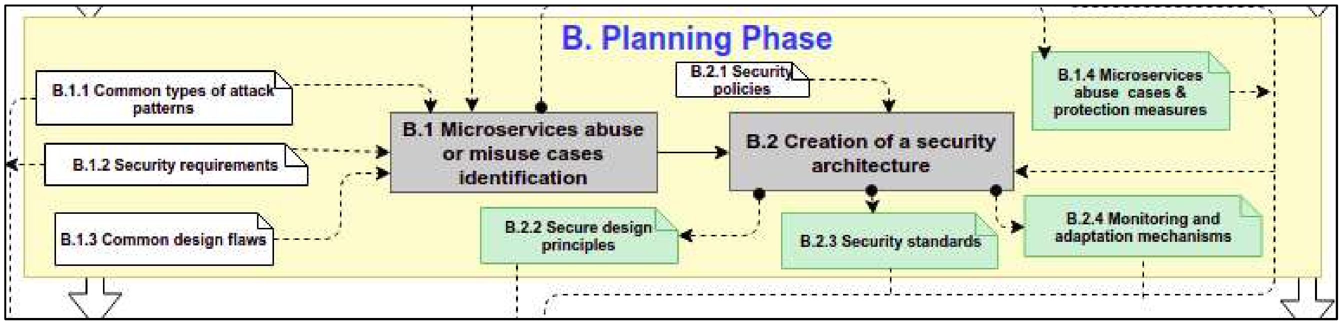

B. Planning Phase
B.1.1 Common types of attack patterns
B.1.2 Security requirements
B.1.3 Common design flaws
B.1 Microservices abuse or misuse cases identification
B.2.1 Security policies
B.2 Creation of a security architecture
B.1.4 Microservices abuse cases & protection measures
B.2.2 Secure design principles
B.2.3 Security standards
B.2.4 Monitoring and adaptation mechanisms

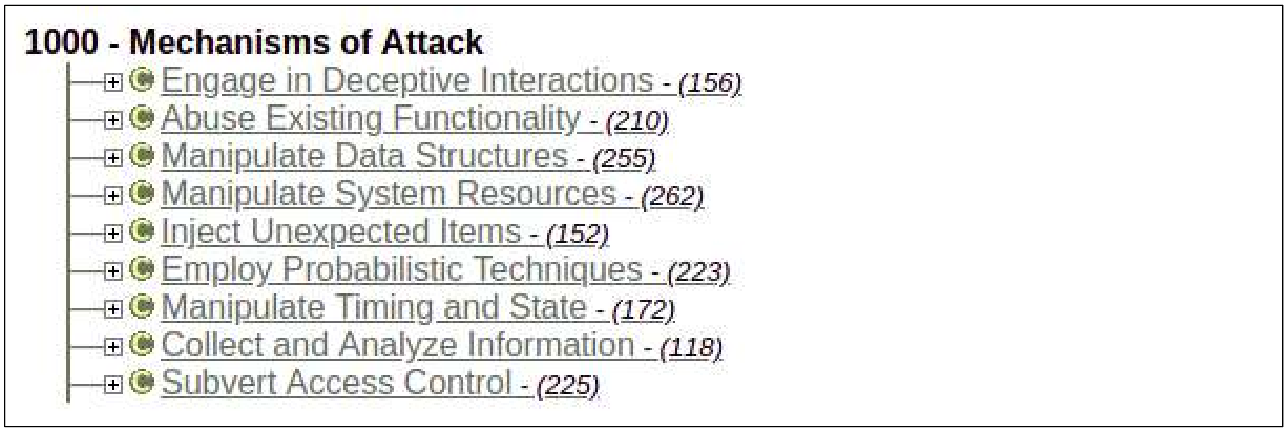
Figure 7.6. SAFEMicroservices planning phase activities

The two security activities are aimed at ensuring that software engineers effectively document the security requirements of microservices composition (SDA-1). These activities are discussed below.

(a) 2.1 Microservices abuse or misuse cases identification

The SAFEMicroservices approach is based on the argument that to secure microservices compositions effectively, software engineers should think like attackers to gain a sense of how an attacker can misuse or abuse the microservices composition. Given the traditional shroud of secrecy surrounding software exploits, SAFEMicroservices takes into consideration that many software engineers are often ill-equipped in software exploitation (Barnum & Sethi 2007). With this in mind, SAFEMicroservices identifies common attack patterns applicable to microservices from the catalog provided by the common types of attack patterns CAPEC (Mitre 2018). An attack pattern describes how an observed attack type is executed (Chrysikos & McGuire 2018). Each attack pattern in the CAPEC catalog is prefixed with *CAPEC* and a number.

The approach adopted by SAFEMicroservices is first to query the CAPEC database using the *mechanisms of attack* search criteria to first identify all the categories of common type of attack patterns. Figure 7.7 below shows the categories of attack patterns under the mechanisms of attack criteria in CAPEC.



1000 - Mechanisms of Attack
- Engage in Deceptive Interactions - (156)
- Abuse Existing Functionality - (210)
- Manipulate Data Structures - (255)
- Manipulate System Resources - (262)
- Inject Unexpected Items - (152)
- Employ Probabilistic Techniques - (223)
- Manipulate Timing and State - (172)
- Collect and Analyze Information - (118)
- Subvert Access Control - (225)

Fig 7.7. Common attack patterns categories in the CAPEC taxonomy

The next step identifies all the attack patterns relevant to each microservices security threat. Table 7.6 gives a non-exhaustive example of a list of attack patterns that are relevant to the *insecure application programming interface* security threat. Associating the CAPEC attack patterns to microservices composition security threat is a manual exercise that requires knowledge gained from analyzing vulnerabilities discussed above.

Fig 7.6 Association of security threats to CAPEC mechanisms of attacks

Security threat	**Applicable CAPEC mechanisms of attack**
Insecure application programming interfaces	• CAPEC-152: Inject unexpected items • CAPEC-210: Abuse existing functionality • CAPEC-255-Manipulate data structures • CAPEC-223: Employ probabilistic techniques

Next, each attack pattern is studied to identify how an attack is executed. An abuse or misuse case is then formulated using the attacker's strategy. Table 7.7 gives examples of abuse or misuse cases created from the *CAPEC- 152: Inject unexpected items* attack patterns. Also, protection measures are identified to mitigate the attack.

Table 7.7. Microservices abuse or misuse case and protection measures

Microservices security threats	**CWE architectural concepts**	**CAPEC mechanisms of attack**	**Abuse or misuse cases**	**Protection measures (Including tools and techniques)**
Insecure application programming interfaces	Validate input	CAPEC: 152: Inject unexpected items	• As an attacker, I can manipulate request parameters to compromise the operation of microservices • As an attacker, I can supply values as parameters to the API that a microservices implementation uses to determine which class to instantiate and I can then create control flow paths through the microservices that were not intended • As an attacker, I can manipulate resource identifiers passed on as parameters to microservices API so that I gain control and perform an action on the resource • As an attacker, I may either alter the path or add/overwrite unexpected parameters in the “query string" on the HTTP query string when calling the microservice REST API • As an attacker, I may supply multiple HTTP parameters with the same name to cause a microservices to interpret values in unanticipated ways • As an attacker, I can exploit a microservices composition component by injecting additional, malicious content during its processing of serialized objects	• Ensure all input content that is delivered to by a microservices is sanitized against an acceptable content specification • Use the validate input secure coding guidelines provided in A.3.1 • Use an input validation framework such as OWASP ESAPI Validation API • Use static analysis tools such as FindBugs on IDE and continuous integrations toolchains to detect input-validation • Perform fuzz testing • Validate object before deserialization process • Limit which class types can be deserialized

SAFEMicroservices also identify tools and techniques as part of the protection measures that software engineers can use to prevent the attacks. Table 7.7 below indicates abuse cases derived for the *insecure application programming interface* security threat and the relevant protection measures. The SAFEMicroservices approach thus enhances the generic CAPEC dictionary into a microservices architecture common attack pattern enumeration and classification *(*MACAPEC*)*. MACAPEC is a vital re-usable security artefact, *security artefact B.1.4*, in SAFEMicroservices that enable software engineers to identify attack patterns associated with each security threat, understand the manner in which the attack is executed and how to protect the microservices composition from attacks. This understanding is essential to software engineers when creating comprehensive security test cases in the coding phase. Security artefacts used by this security activity are listed below.

Required Security Artefacts:

A.1.1 Security threats and associated vulnerabilities

B.1.1 Common types of attack patterns

B.1.2 Security requirements

B.1.3 Common design flaws

B.1.4 Microservices abuse cases & protection measures

The last step is to ensure that identified protection measures comprehensively cover all the known common design flaws. A gap analysis is performed using the top ten known and common mistakes that software engineers make when designing software as provided for example, by the IEEE (2015). The list of common design mistakes is reviewed to see if each flaw has been addressed by the protection measures identified above. Protection measures are identified for any flaws not addressed using the CAPEC catalog. Table 7.8 is an example of the outcome of the gap analysis.

Table 7.8. Gap analysis of protection measures for common design flaws

Common design flaws	**Protection measures**
Earn or give, but never assume trust	• Use protection measures defined for CAPEC: 152- Inject unexpected items
Use an authentication mechanism that cannot be bypassed or tampered with	• Use multi-factor authentication • Use time-tested authentication mechanisms • Authentication system designs should automatically provide a mechanism requiring re-authentication after a period of inactivity or before critical operations
Authorize after you authenticate	• Use defense in depth strategies
Strictly separate data and control instructions, and never process control instructions received from untrusted sources	• Use protection measures defined for CAPEC: 152- Inject unexpected items

The next step after identifying the microservices abuse cases and protection measures is to create a security architecture of a microservices composition, discussed next.

(b) 2.2 Creation of a security architecture

A security architecture is an abstraction of a design that identifies and describes where and how security controls are used (Maikel & Asim 2018). In SAFEMicroservices the aim of creating a security architecture is to enable software engineers to quickly design and create secure microservices composition using reusable building blocks. SAFEMicroservices assumes the existence of security policies within an organisation that provide high-level information security goals within an organisation the security.

The security architecture in SAFEMicroservices is based on the balance and holistic mix of the following elements:

Required Security Artefacts:

A.3.1 Catalog of architectural-level secure coding guidelines

B.2.1 Security policies and standards

B.2.2 Secure design principles

B.2.3 Monitoring and adaptation mechanisms

Indirectly, the architecture-centric threat modeling performed in the preliminary phase and the security awareness derived from an analysis of security threats and vulnerabilities and the identification of abuse cases influences this step. These security artefacts are described next.

Secure design principles

A principle can be defined as a qualitative statement of intent that should be met by the architecture (Maikel & Asim 2018). Secure design principles are vital in SAFEMicroservices because they establish the basis for a set of design rules for microservices and also influence the implementation of security controls. In SAFEMicroservices these principles are elicited mostly from analysing and grouping of CWE vulnerabilities according to a common theme. Table 7.9 below shows the secure design principles for microservices elicited from CWE vulnerabilities that share the common theme. A complete list of secure design principles is provided in Appendix B, *security artefact B.2.2.* Table 7.9 can be used as a cheat sheet of design principles to apply and the relevant vulnerabilities that software engineers should avoid.

Table 7.9. SAFEMicroservices secure design principles

CWE vulnerabilities	**Security designs principles**	**Principle description**
CWE-272: Least privilege violation CWE-250: Execution with unnecessary privileges	The principle of least privilege	All components in a microservice composition should be assigned minimum necessary rights when accessing any resource, and the rights should be in effect for the shortest duration necessary.
CWE-636: Not failing securely	The principle of failing securely	In the event of a component in a microservices composition failing, it should do so securely
CWE-656: Reliance on security through obscurity	The principle of defense in depth	The components should use layering of security defenses to reduce the chance of a successful attack
CWE-637: Unnecessary complexity in the protection mechanism	The principle of economy of mechanism	The components should ensure that multiple conditions are met before granting access permission
CWE-269: Improper privilege management CWE-268: Privilege chaining	The principle of separation of privilege	The design of each component should be kept simple

Next, the creation of security standards is discussed.

Microservices security standards

Standards are directives that establish mandatory mechanisms that software engineers must comply with. Security standards in SAFEMicroservices ensures that software engineers to develop microservices that can resist, detect, react and recover from any attack. The security standards are formulated to avoid:

- known security threats and vulnerabilities discussed.
- known attack patterns discussed above.

Table 7.10 below gives a non-exhaustive list of security standards to guide the development of microservices. A comprehensive list is provided in Appendix B *security artefact B.2.3*. Other directives should be derived from the organization security policies.

Table 7.10. Microservices security standards

Microservices Security Standards
Any client communication with a microservice must be done via API Gateway to provide load balancing, and a standard set of security capabilities and communication to API gateway should be authenticated
Each microservice must be protected using a defense in depth approach
The microservices composition must use a well-known and secure open standard protocol for centralized authentication using tokens. The token must be generated using an algorithm that follows the cryptography standard and should have an associated time to live
Authentication tokens must be encrypted
Each microservices must have a unique API key for calling another microservice
API calls made by users and microservices must be limited to only those necessary for those users or microservices to perform their functions
All API requests must be logged to a centralized logging and monitoring system
A tool to monitor and visualize inter-microservice communication must be deployed as part of the management capabilities of the microservices architecture
All communication in the microservices composition must use transport layer security
All microservices must run in an approved application container technology
Containers must be configured according to approved security best practices

The secure design principles and the security standards ensures that microservices composition resist attacks. However, microservices compositions should also be designed to be able to detect attacks, react to attacks and recover from attacks. In this regard, microservices should be built with both monitoring and adaptation in mind. The next section discuss how software engineers can be equipped to design such microservices.

Monitoring and adaptation mechanisms

SAFEMicroservices' aim to monitoring is twofold. Firstly, microservices need to be built with appropriate logging of user activities, and monitoring tools are needed to monitor microservices. Secondly, software engineers should be guided to avoid vulnerabilities that can results from improper implementation of logging, such as logging of sensitive information. With this in mind, SAFEMicroservices uses common weakness types from CWE associated with logging and formulate recommendations to guide software engineers. Figure 7.11 below shows an example of monitoring guidelines. A comprehensive list of monitoring guidelines is provided in Appendix B *security artefact B.2.4.*

Table 7.11. Microservices monitoring guidelines

Monitoring guidelines	**CWE Vulnerabilities**
Log all information important for identifying the source or nature of an attack	CWE-223: Omission of Security-relevant Information
Do not log sensitive information on the log	CWE-532: Information Exposure Through Log Files
Log information in much details	CWE-778: Insufficient Logging

Software engineers are free to use monitoring tools of their choice. SAFEMicroservices recommends that software engineers adopt approaches such as the circuit breaker pattern and load balancing to ensure microservices respond appropriately to attacks at runtime as adaptation mechanisms.

(c) Summary of planning phase deliverables

The SAFEMicroservices planning phases produces four reusable security artefacts as shown in Table 7.12.

Table 7.12. Summary of planning phase deliverables

Planning phase deliverables	**Summary of deliverable**
B.1.4 Microservices abuse and misuse cases	Catalog of strategies attacker can use to exploit microservices
B.2.2 Security standards	Directives software engineers should comply with to ensure secure microservices
B.2.3 Secure design principles	Guideline to be followed to ensure secure microservices composition
B.2.4 Monitoring and adaptation mechanisms	Guidelines to ensure microservices are monitored and to ensure microservices can respond to attacks

The next section discusses the activities of the coding phase.

7.2.3 Coding phase

The coding phase in SAFEMicroservices commences when software engineers use the various security artefacts from the preliminary phase and planning phase to write secure microservices source code. Figure 7.8 below shows the five essential security-focused activities of SAFEMicroservices coding phase.

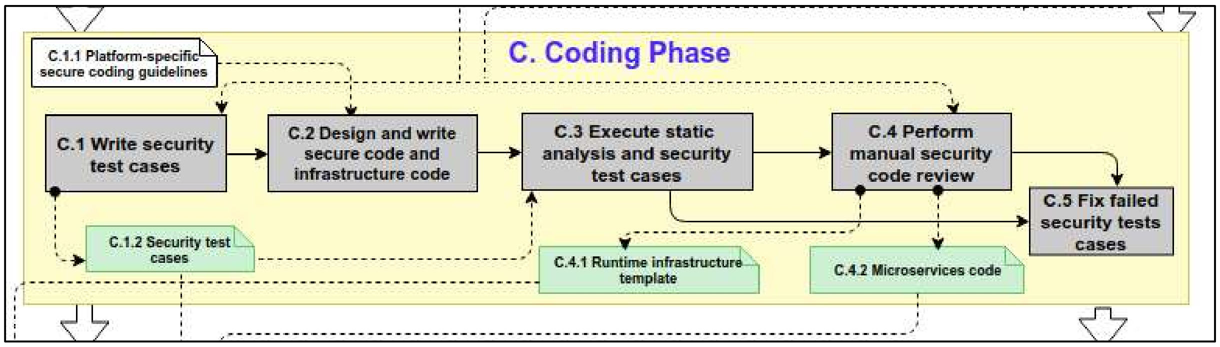

C. Coding Phase
C.1.1 Platform-specific secure coding guidelines
C.1 Write security test cases
C.2 Design and write secure code and infrastructure code
C.3 Execute static analysis and security test cases
C.4 Perform manual security code review
C.5 Fix failed security tests cases
C.1.2 Security test cases
C.4.1 Runtime infrastructure template
C.4.2 Microservices code

Figure 7.8. SAFEMicroservices coding phase activities

The five essential security focused-activities of the coding phase shown in Figure 7.8 above can be viewed as secondary activities that aim to address the need adopt secure programming best practises (SDA-2) and validate security requirements and secure coding standards (SDA-3). First, the SAFEMicroservices artefacts required to perform the five activities comprehensively are shown next, as well as all resultant security artefacts.

Required Security Artefacts:

A.3.1 Catalog of architectural-level secure coding guidelines
A.4.2 Secure pipeline
B.1.4 Microservices abuse cases and protection measures
B.2.1 Security policies and standards
B.2.2 Secure design principles
C.1.1 Platform-specific secure coding guidelines
C.1.2 Security test cases
C.4.1 Runtime infrastructure template
C.4.2 Microservices code

The relationship between the various security artefacts is vital to understand SAFEMicroservices. To this end, Figure 7.9 below presents a conceptual model that depicts in brief the relationship between these artefacts. The security goal specifies the security capability of a microservices composition and the security policy states what protection mechanisms need to be implemented on a microservices.

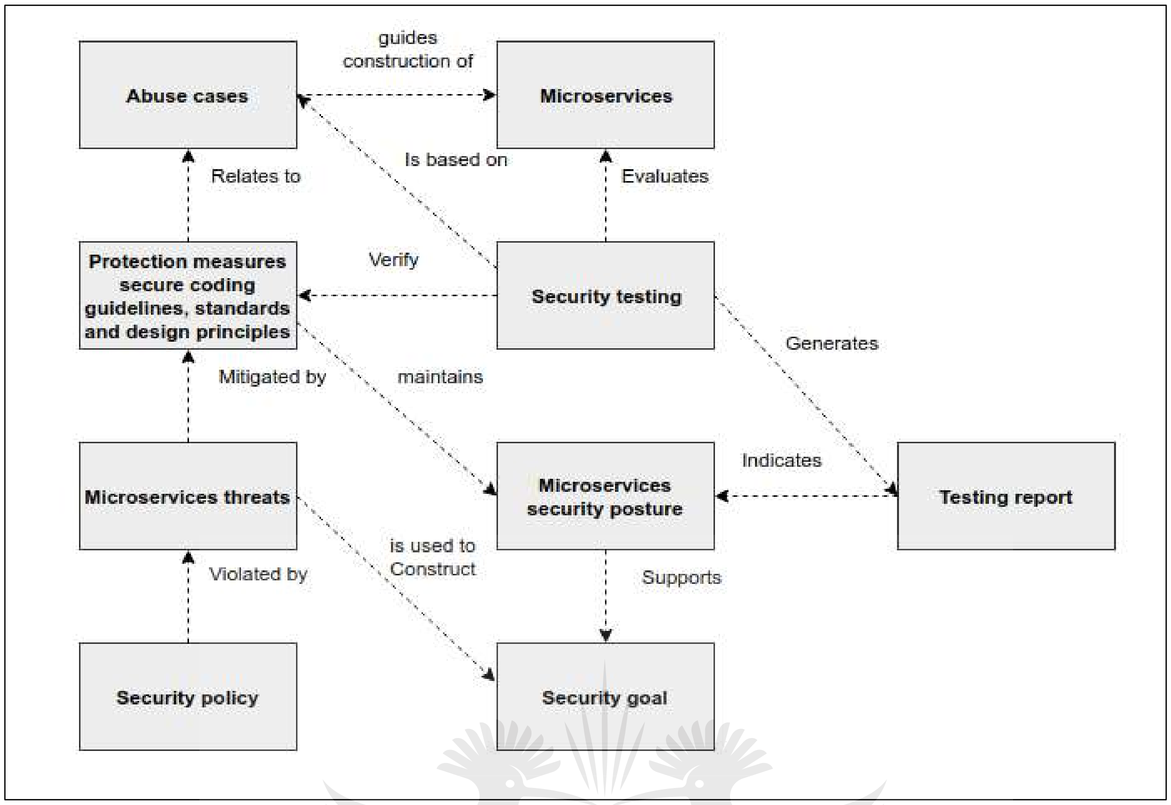


Figure 7.9. Conceptual model of relationship between SAFEMicroservices artefacts

Next, the five activities of the coding phase are discussed.

(a) C.1 Write security test cases

SAFEMicroservices requires that software engineers write both *security unit test* cases and *acceptance tests cases*. The purpose of the security unit test cases is to validate an individual unit that makes up a microservices to determine if each unit meets the security expectation. The acceptance test cases determine whether the microservice or the microservices composition satisfies a given security criterion during its operation. The security and acceptance test cases should be written in such a way that any violation of security artefacts A.3.1, B.1.4, B.2.1, B.2.2 and C.1.1 are detected. In SAFEMicroservices, software engineers are free to use libraries or technologies of their choice to write security test cases. The security test cases should be written both for microservices and the runtime infrastructure and should be comprehensive to ensure extensive test coverage of the written source code.

(b) C.2 Design and write secure microservices code and infrastructure code

Software developers write secure code for both microservices and templates for automating the creation of the deployment infrastructure. Software engineers need to ensure that the design and writing of code follow the guidelines specified in security artefact A.3.1, B.2.1, B.2.2 and C.1.1 and that protection measures are incorporated into the microservices. In addition, microservices should be designed and developed in such a way that they are fully equipped to respond to attacks at runtime using the adaptation mechanisms that are built from adaptation requirements discussed in the security architecture of the composition. The microservices should also be built to support comprehensive monitoring as discussed before.

SAFEMicroservices adopts the concept of "infrastructure as code". Infrastructure as code manages and provisions deployment infrastructure using source code templates that are executed by configuration tools (Morris, 2016). Using template to create infrastructure, enables software engineers to test templates using security test cases. This ensures that deployment infrastructure is thoroughly tested before deploying any microservices. The deployment infrastructure should be created with minimum operating system services and network functionality. The aim is to promote microservice that are secure-by-default. Changes to the infrastructure template should be versioned, tested and tracked to ensure auditability.

(c) C.3 Execute static analysis and security test cases

Software engineers need to test the code for both microservices and the infrastructure before the changes are submitted to the shared repository. Testing should utilize the static analysis tools installed on the software engineer's integrated development environments (IDE). In addition, before committing source code to a shared repository, a software engineer should ensure that all the security test cases have been executed successfully.

(d) C.4 Perform manual security code reviews

Manual security code reviews ensure that software engineers collaborate to create safe microservices. Reviews are vital to identifying some of the design flaws that cannot be identified by tools and also to identify security flaws such as hardcoded credentials. In addition, manual reviews are vital to encourage software engineers to write readable microservices source code that is easy to maintain.

(e) C.5 Fix failed security tests cases

Software engineers need to address any security vulnerabilities before submitting their source code to a common repository. By so doing, software engineers receive early feedback on security issues, and security vulnerabilities can be given early attention in the microservices development life-cycle.

(f) Summary of the coding phase deliverables

The SAFEMicroservices security artefacts produced by the coding phase is a suite of security test cases, a tested template to create the runtime infrastructure and source code for microservices. These artefacts are reusable components in the development of secure microservices. The activities of the coding phase ensure that the objective of SAFEMicroservices of ensuring that software engineers get timely feedback on security vulnerabilities on the microservices source code is met.

Next, the code integration phase is discussed.

7.2.4 Code Integration phase

The code integration phase in SAFEMicroservices commences when the software changes made by software engineers have been successfully tested by a software engineer on the IDE, and the software changes have also successfully been peer-reviewed. The objective of SAFEMicroservices is to automate the building and packaging of software changes, the execution of a suite of test cases, and validation of microservices artefacts using the development infrastructure created in the preliminary phase. Figure 7.10 below shows the three essential security-focused activities of the SAFEMicroservices coding integration phase extracted from the coding integration phase in Figure 7.3.

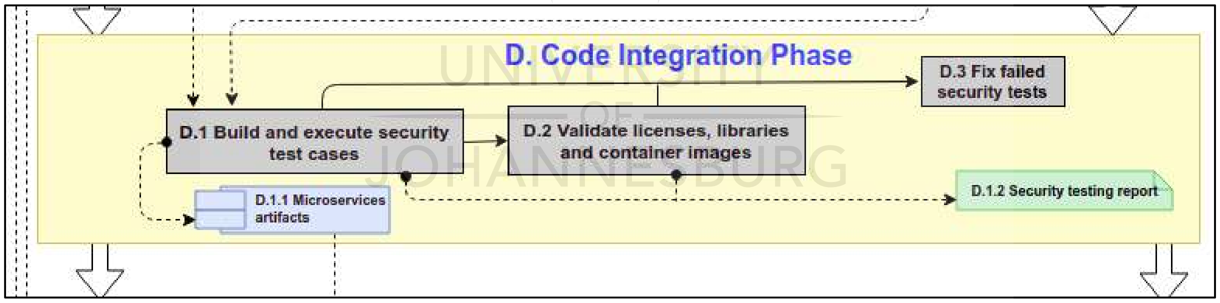

D. Code Integration Phase
D.1 Build and execute security test cases
D.2 Validate licenses, libraries and container images
D.3 Fix failed security tests
D.1.1 Microservices artifacts
D.1.2 Security testing report

Fig 7.10. SAFEMicroservices code integration phase activities

The three essential security-oriented activities of the coding integration phase shown on Figure 7.10 aim to assist software engineers to adopt secure programming best practices (SDA-2) and to validate security requirements and secure programming best practices (SDA-3). These three activities are discussed next.

(a) D.1 Build microservices and execute security test cases

The building of microservices source code is performed by code integration tools. Software engineers are free to choose the code integration tool of their choice, preferably the tool should build and execute test cases in an automated manner. The build process should produce deployable microservices artefacts and execute test cases written by the software engineer in the coding phase.

(b) D.2 Validate licenses, libraries and container images

In SAFEMicroservices, it is vital to ensure that any open source software libraries used in microservices are known and any security vulnerabilities in those libraries are detected. Also, the third-party libraries should be from trusted sources. This activity is vital to ensure that microservices compositions do not inherit security vulnerabilities from third-party components. The code integration system should provide mechanisms to validate third-party libraries and their respective licenses. In addition, the code integration system should also validate container images. The validation process should produce a security testing report which is used to decide if the software changes are safe enough to be deployed to production. The security testing report is a vital artefact that should be used to sign-off microservices for deployment in both the pre-production and the production environment.

(c) D.3 Fix failed security test cases

Software engineers need to attend to any security vulnerabilities that may be detected during the execution of the test cases by the build tool or the validation of third-party libraries and container images.

(d) Summary of the code integration phase deliverables

The deliverables of the code integration phase are the microservices artefacts that are ready to be deployed into a pre-production environment and a security testing report that provide software engineers with a view into the security state of the microservices artefacts, the third-party libraries used and the security status of the container images. The activities of the code integration phase ensure that the objective of SAFEMicroservices of making sure that software engineers are offered guidance on how to incorporate security-oriented activities, tools and techniques in their daily software development tasks is met. Furthermore, the activities also assist software engineers to get timely feedback on security vulnerabilities on the microservices source code before any deployment.

7.2.5 Pre-production deployment phase

The pre-production phase in SAFEMicroservices is used to deploy microservices artefacts to an environment where business users can perform quality assurance before a decision is made to deploy the microservices in a production environment. Figure 7.11 below shows the security-focused activities of the SAFEMicroservices pre-production deployment phase. Figure 7.11 is an extract of the pre-production deployment phase shown in Figure 7.3 above.

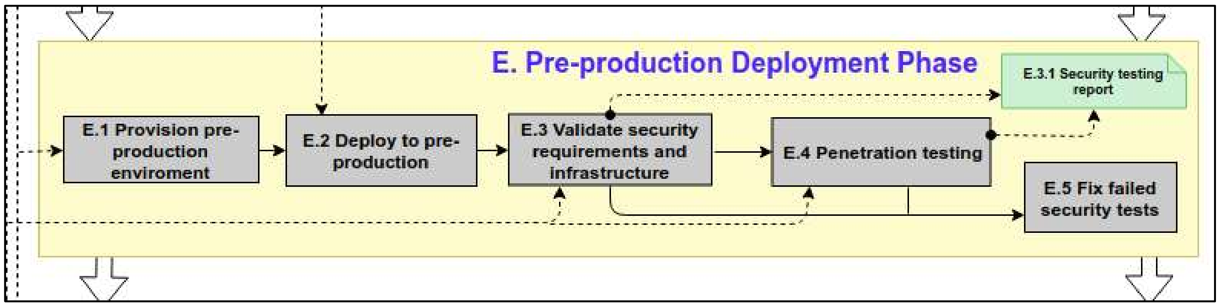

E. Pre-production Deployment Phase
E.3.1 Security testing report
E.1 Provision pre-production enviroment
E.2 Deploy to pre-production
E.3 Validate security requirements and infrastructure
E.4 Penetration testing
E.5 Fix failed security tests

Fig 7.11. SAFEMicroservices pre-production deployment phase activities

The essential five security focused-activities of the pre-production deployment phase shown on Figure 7.11 aims to assist in validating security requirements and secure programming best practices (SDA-3) and to ensure the secure configuration of the runtime infrastructure (SDA-4). These five activities are discussed next.

(a) E.1 Provision pre-production environment

The provisioning of a pre-production environment should be automated as much as possible in SAFEMicroservices. A configuration management tool should be used to create the infrastructure in conjunction with the infrastructure template, an artefact created and tested in the coding phase. The use of templates ensures that consistent configurations are created that satisfy the security requirements. This goes a long way to ensure secure-by-deployment in microservices. The environment should be validated to ensure for example that no unnecessary services are running that can compromise security.

(b) E.2 Deploy to pre-production

Once the environment has been created and validated the microservices can be deployed. Software engineers can use tools of their choice to deploy microservices. However, the deployment pipeline should be secured to ensure that an attacker does not gain access and deploy corrupted microservices artefacts.

(c) E.3 Validate microservices

Software testers manually test the microservices and ensure that both functional and non-functional requirements of the microservices are satisfied. In addition, there is a need to make sure that microservices' access to certain resources such as file and directory is only limited to what is necessary for microservices to perform their function.

(e) E.4 Penetration testing

The last step of the pre-production deployment phase is to ensure that penetration testing is performed. Penetration testing is an authorized simulated attack on microservices composition performed to evaluate security. Penetration testing is performed to identify vulnerabilities, including the potential for unauthorized parties to gain access to microservices and data. SAFEMicroservices does not dictate which tools to use to perform the test, and software engineers can use any tools of their choice. What is essential in SAFEMicroservices is that feedback from penetration testing is given attention before microservices are deployed to a production environment.

(f) E.5 Fix failed security tests

The purpose of this activity is to ensure that software engineers address any security vulnerabilities that may be detected during the penetration testing. This is vital to ensure that microservices are not deployed with known security vulnerabilities.

(g) Summary of pre-production deployment phase deliverables

The essential SAFEMicroservices artefact of the pre-production deployment phase is a security testing report that covers both the results of security requirements validation done manually by software testers and also the outcome of the penetration testing exercise. The stakeholders that have a vested interest in the application can use both results from functional requirements validation together with the security testing report to decide if the microservices can be deployed into a production environment. The stakeholder can analyse the security testing report and decide the priority of any reported vulnerabilities. If the vulnerabilities are of a low priority, then a decision to deploy the microservices can be made.

7.2.6 Operational phase

SAFEMicroservices operational phase aims to ensure that microservices compositions always maintain their security posture while in use. Any microservices maintenance activities should not degrade the protection measures of the microservices composition. Figure 7.12 below shows the three essential security-focused activities of SAFEMicroservices operational phase extracted from Figure 7.3 above.

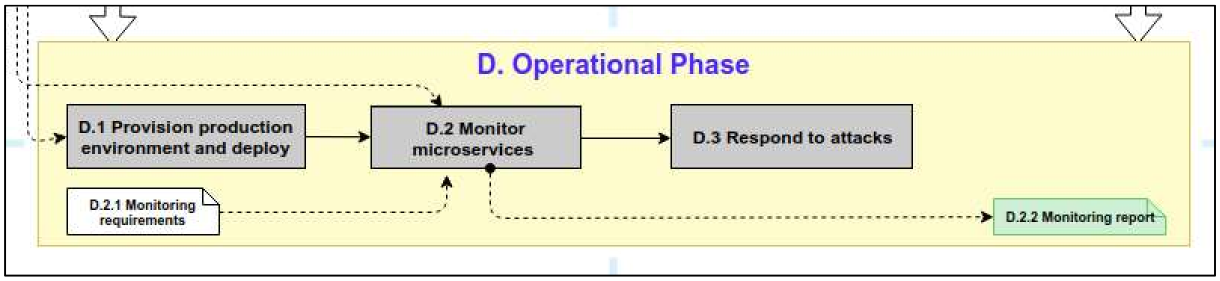

D. Operational Phase
D.1 Provision production environment and deploy
D.2 Monitor microservices
D.3 Respond to attacks
D.2.1 Monitoring requirements
D.2.2 Monitoring report

Figure 7.12. SAFEMicroservices operational phase activities

The objective of the three essential security focused-activities of the operational phase shown in Figure 7.12 is to assist software engineers to continuously monitor the behaviour of components of the microservices composition (SDA-4) and to ensure that a microservices composition securely respond to attacks using adaptation mechanisms (SDA-6) at runtime. These three essential activities are discussed next.

(a) D.1 Provision production environment and deploy

The first activity is to create a secure microservice runtime environment. This environment should be created using a tested infrastructure template, that was used to create the pre-production and went through rigorous testing using penetration testing techniques. This ensures that a safe runtime environment

(b) D.2 Monitor microservices composition

SAFEMicroservices requires that software engineers constantly gain access to the behaviour of the microservices composition. In this regards, various tools should be used to monitor microservices. As discussed in the coding phase, microservices should be built with an inherent ability to trace user activities in order to identify attackers and any malicious modifications. Monitoring requirements that are part of the security architecture should guide software engineers in ensuring that both the monitoring infrastructure is in place and microservices are comprehensively monitored to detect any attacks at runtime.

(c) D.3 Respond to attacks

Microservices should be able to respond securely to attacks using adaptation mechanisms that are inbuilt within the microservices. The adaptation mechanisms are part of the security architecture and were discussed in the preliminary phase.

(d) Summary of the operational phase deliverables

The operational phase in SAFEMicroservices should provide a real-time monitoring view of microservices. Various tools reviewed in Chapter 6 can be used to provide a real-time dashboard that software engineers can use to gain continuous insight into the operation of microservices.

The next section discusses the benefit of SAFEMicroservices framework.

7.3 Summary of SAFEMicroservices benefits

Table 7.13 below provides a summary of the expected benefits of using the SAFEMicroservices approach in each phase.

Table 7.13. SAFEMicroservices benefits

Phase	**SAFEMicroservices benefits**
Preliminary phase	• Promotes security awareness among software engineers and other stakeholders so that software security is accommodated in any technology migration plan • Software engineers proactively understand and identify threats and potential vulnerabilities early in the development process. This helps to mitigate potential design flaws that are usually not easily found using other techniques such as code reviews and static source analysis • Software engineers gain an opportunity to fix security vulnerabilities early in the design phase and avoid expensive re-engineering efforts that may be required after source code is written or a security breach has occurred
Planning phase	• Software engineers use abuse or misuse cases to devise upfront defense mechanisms that cover all possible microservices attack pattern • Software engineers use abuse or misuse cases to create a comprehensive roadmap for security testing of microservices
Coding phase	• Software engineers are responsible for secure development • Security vulnerabilities like buffer overflow, SQL injection and cross-site scripting can be identified by static analysis tools and given early attention • An improvement of the developer's security knowledge using static analyzers tools that provide suggested security corrections and improvements to the code • Software engineers get timely feedback on violation of secure coding guidelines • Promotion of collaboration and sharing of knowledge on software quality among software engineers • Software engineers are encouraged to write readable software code using code review mechanism
Code integration phase	• Ensure that software engineers get timely feedback on security vulnerabilities on the software source code • Automate security testing

Pre-production deployment phase	• Feedback from penetration testing is easily integrated into the software development process
Operational phase	• Software engineers get a continuous insight into the operation of software in a production environment.

Table 7.14 below gives a not exhaustive list of tools that software engineers can use in SAFEMicroservices as reviewed in Chapter 6. Software engineers are free to choose any tools of their choice.

Table 7.14. Example of tools for SAFEMicroservices

Phase	**Tool or Method Classification**	**Example of Required Tools and Techniques**
Preliminary phase	Threat modeling tools	Threat Dragon (OWASP 2018), STRIDE, Attack trees (Saini, Duan & Paruchuri, 2008), misuse cases (Sindre & Opdahl, 2005)
Planning phase	Security planning tools and secure designs	SAFEcode Security user-stories (ben Othmane, Angin & Bhargava, 2014,), OWASP Application Security Verification Standard (Boberski, Williams & Wichers, 2009)
Coding phase	Static analysis tools	FindSecurityBugs (Arteau, 2016), Brakeman (Collins, 2012), SonarQube (Guaman et al., 2017), Xanitizer, (Xanitizer, 2017), VisualCodeGepper (Alsmadi et al., 2018)
	Code review tools	Crucible (Rigby et al., 2012), Collaborator (Wang et al., 2012) and Gerrit (McIntosh et al., 2016).
Code integration phase	Continuous integration tools	Jenkins (Soni & Berg, 2017), TeamCity (Mahalingam, 2014), Bamboo (Watson, 2016), GitLab (Cheng, 2017), Travis (Travis, 2015)
	Open source license checker	Whitesource (Harutyunyan, Bauer & Riehle, 2018), Open source License checker (Kapitsaki, Kramer & Tselikas, 2017)
Pre-production phase	Environment configuration tools	Chef (Taylor & Vargo, 2014), Puppet (Loope, 2011), Ansible (Hall, 2015)
	Container validation tools	Anchore (Anchore, 2018), Clair (Clair, 2018), Docker Bench (Tak et al., 2017)
Operational phase	Monitoring tools	appDynamics, Dynatrace and Prometheus
	Response to attacks patterns	circuit breaker pattern (Fontesi & Weber, 2016), Netflix Hystrix (Christensen, 2012)

Next, a conclusion is provided.

7.4. Conclusion

The development of a secure microservices composition that can continue to function securely under malicious attacks is a complex exercise. In this regard, the SAFEMicroservices framework provides a methodology that can address this challenge successfully. SAFEMicroservices provides a coordinated approach to assist software engineers to implement and manage microservices security controls effectively. SAFEMicroservices offers a holistic approach to security and identifies opportunities in the software development life-cycle where security-focused tools and techniques can be leveraged.

The essential contribution of SAFEMicroservices is a systematic and flexible approach to security that accommodates variations in the implementation using different technologies and the risk profile of each microservices composition. SAFEMicroservices makes the development of secure microservices a part of the software development culture. The systematic integration of security testing further reinforces the secure software development culture into all phases of the software development life-cycle that ultimately improves the software security skills of software engineers.

In the next chapter, SAFEMicroservices is validated using selected security-focused tools and techniques. A microservices composition is designed and developed using the SAFEMicroservices approach to demonstrate that SAFEMicroservices can be used to develop secure microservices.

Chapter 8

SAFEMicroservices Framework Implementation

8.0 Introduction

Chapter 7 proposed and discussed SAFEMicroservices, a practice-oriented framework to assist software engineers to develop secure microservices. The security-oriented activities in SAFEMicroservices assist software engineers to use suitable tools and techniques during their daily microservices development tasks. The next step is to perform an empirical evaluation of SAFEMicroservices to observe by means of an instantiation, if the framework is adequately specified to support the development of secure microservices compositions.

Accordingly, this chapter discusses the use of SAFEMicroservices to develop PickMeUp, a microservices composition discussed in Chapter 3. The security-oriented activities of SAFEMicroservices are used to identify and refine protection measures and ensure their integration into PickMeUp to ensure an application that can resist, detect and respond to attacks. SAFEMicroservices is also used at various development stages of PickMeUp to ensure traceability of analysis, design, coding, and testing of the microservices. SAFEMicroservices is used in such a way that the development of PickMeUp provides a proof-of-concept of the framework.

The chapter is organized as follows: Section 8.1 defines the evaluation criteria used to determine the success of the implementation in this chapter. Section 8.2 provides an overview of PickMeUp. The technologies chosen to develop PickMeUp are discussed in Section 8.3. Section 8.4 discusses the software development methodology adopted to develop PickMeUp. Section 8.5 explains the inception stage of PickMeUp and how SAFEMicroservices is used to ensure a security risk assessment. Section 8.6 discusses how SAFEMicroservices is used in the construction of various architecture component of PickMeUp. In Section 8.7, integration testing and hardening of features of PickMeUp that are developed incrementally are discussed. Section 8.8 discusses the deployment and monitoring of PickMeUp. Section 8.9 discuss the evaluation results. A conclusion then follows in Section 8.10.

8.1 Evaluation criteria

The objective of the implementation in this chapter was to determine if SAFEMicroservices provided the following:

- Easy to follow and effective steps – the researcher considered the ease of use as an ideal attribute considering that software engineers are under pressure to delivery software. The framework should therefore not impact the productivity of engineers. Besides, the value of the framework is in its effectiveness in ensuring secure software.
- Easy support of security-focused tools – the researcher considered tools support as a success criterion because as discussed in Chapter 6, there is limited guidance on how tools can be integrated to assist software engineers create secure applications. In this regard, freely available tools and technologies are used throughout the development of PickMeUp.
- Easy with which the framework can be used in an iterative software development method - the aim of this criteria was to determine if the proposed framework could be used with a software methodology that is used mostly in industry.

The above three attributes were considered the three-fold basic success criteria of the SAFEMicroservices by the researcher.

8.2 Overview of microservices composition for prototyping

PickMeUp is an imaginary on-demand taxi application such as Uber (Cramer & Krueger, 2016). The PickMeUp microservices composition was introduced in Chapter 3. PickMeUp was decomposed into a set of microservices by first identifying the functional business capabilities. The scope or functional context of each microservice was determined using the single responsibility principle (SRP), and the common closure principle (CCP) discussed in Chapter 3. Figure 8.1 below provides a high-level architecture diagram showing the interaction of various components. The user interface is not considered part of the microservices composition in this discussion.

The microservices responsible for handling business functionality are defined below:

- Trip management microservice – a service that handles all requests for a trip from a passenger.
- Passenger management microservice – a service for managing passenger information.
- Driver management microservice – a service for managing driver information.
- Passenger notification microservice - a service responsible for all forms of passenger notifications.
- Driver notification microservice – a service responsible for all forms of driver notifications.
- Payments microservice – a service that handles all forms of payments for service rendered.

PickMeUp also has other microservices dictated by the microservices architectural style and non-functional requirements such as authentication. These microservices are defined below:

- API gateway – an entry point into the microservices composition for all external clients.
- Service registry – a registry to enable microservices in PickMeUp to locate one another.
- Message broker – a message buffer to allow microservices to communicate by sending messages when necessary.
- Authentication microservices – a service responsible for security.

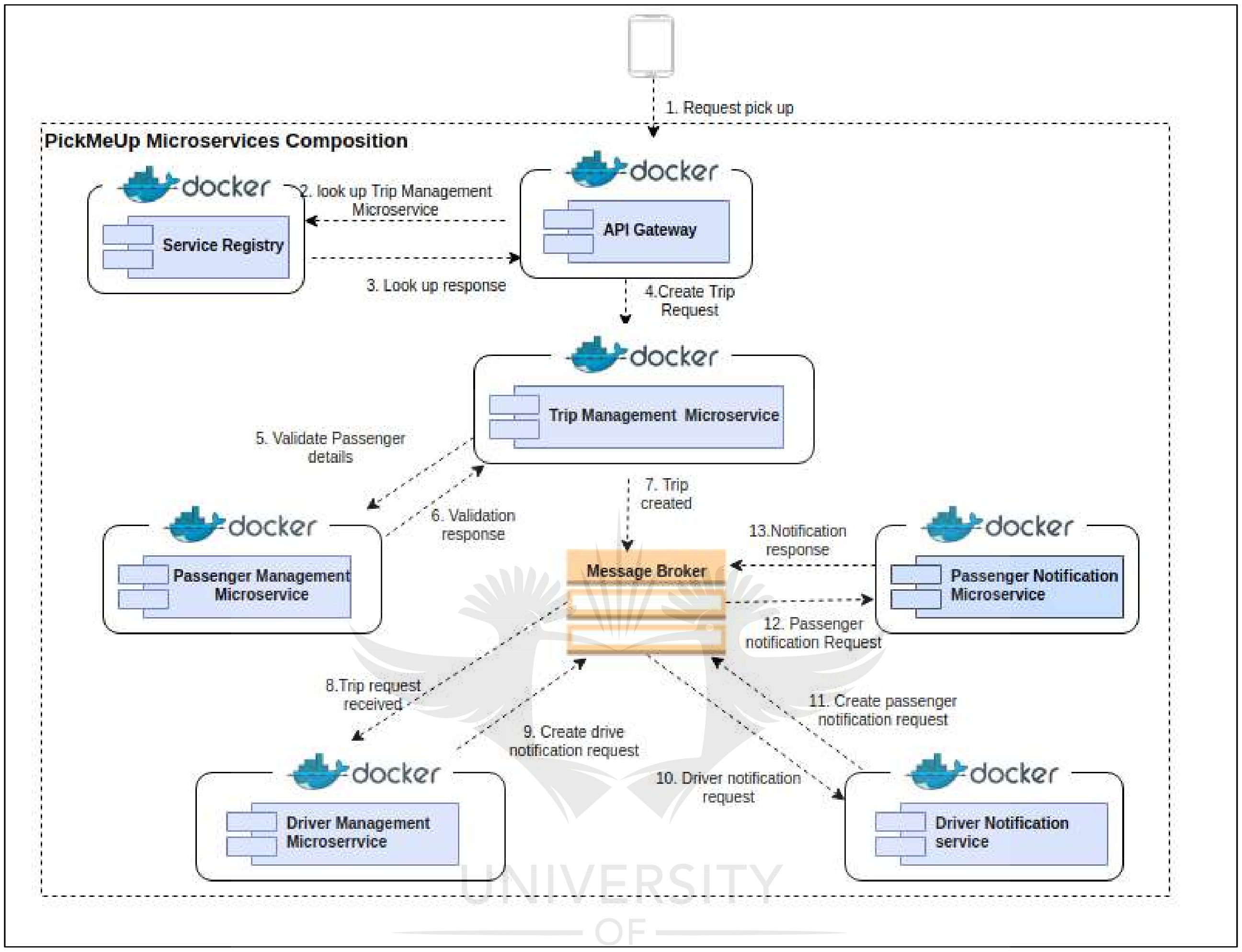

1. Request pick up
PickMeUp Microservices Composition
docker
Service Registry
2. look up Trip Management Microservice
3. Look up response
API Gateway
4.Create Trip Request
Trip Management Microservice
5. Validate Passenger details
6. Validation response
Passenger Management Microservice
7. Trip created
Message Broker
13.Notification response
12. Passenger notification Request
Passenger Notification Microservice
8.Trip request received
9. Create drive notification request
10. Driver notification request
11. Create passenger notification request
Driver Management Microserrvice
Driver Notification service

Figure 8.1. PickMeUp Microservices composition

The next section discusses the technologies chosen to implement PickMeUp.

8.3 Implementation technologies

The Spring Boot framework (Gutierrez, 2016) was chosen as a technology to develop PickMeUp microservices. Spring Boot is a readily available Java-based technology framework that simplifies the development of RESTful web services using the microservices architectural style. Various freely and readily available software libraries were used in conjunction with the Spring Boot technology framework to create the components of PickMeUp. The libraries are identified below:

- Microservices - the RESTful communication model using the JSON messaging format was used to develop the PickMeUp microservices. Each microservice exposed a RESTful API using the Spring Framework libraries (Varanasi & Belida, 2015).
- The API gateway - the API gateway was developed using the Netflix Zuul library (Netflix, 2019). The Netflix Zuul library is a freely available library that easily integrates into the Spring Boot framework.
- Service registry - the service registry was developed using the Eureka library (Netflix, 2019). Eureka is a REST-based service that is primarily used for locating microservices for load balancing and failover (Netflix, 2019). The client-side discovery approach was used as a mechanism for microservices to locate one another by directly querying the Eureka-based service registry.
- Message broker - the message broker was developed using RabbitMQ (Videla & Williams, 2012). RabbitMQ is easily supported by the Spring Boot framework.
- PostgreSQL (Obe & Hsu, 2017) database was used to store data.
- Docker was used as the container technology to deploy each component of PickMeUp.

8.4 Software development methodology

SAFEMicroservices supports both sequential and iterative software development methodologies. The sequential software development approach suits the integration of security-focused activities, and the use of security validation strategies between the analysis, design, coding, and testing development stages. On the other hand, the iterative and incremental software development methodologies limit their ability to accommodate the security-focused activities and the use of security validation strategies (ben Othmane et al., 2014). With this in mind, an Agile methodology, an iterative approach was chosen to develop PickMeUp to determine the suitability of SAFEMicroservices in an iterative software methodology and in the process add value to the research in this study.

As SAFEMicroservices was integrated with an Agile methodology, it needed to be identified where security checkpoints would fit in. An understanding of the Agile methodology was therefore required. The Agile software development methodology can be divided into three stages namely (Ambler, 2013):

- Inception stage – defines the scope of the project and model of the initial architecture.
- Construction stage – develops the software in a set of iterations.
- Transition stage – the stage is used for integration testing and hardening the software deliverable developed in iterations to make it ready as a release for use in a production environment.

Figure 8.2 below shows how the SAFEMicroservices security checkpoints were positioned in the Agile software development methodology. The operational phase of SAFEMicroservices was considered beyond the scope of the Agile software development methodology.

The Agile methodology capture a functional requirement as a *user story.* A *user story* is a way to capture a description of a software feature from an end-user perspective (Beck et al., 2001). The user stories are added to a list called *product backlog.* The software features encapsulated in user stories are developed incrementally in the construction stage and comprehensively tested in the transition stage to ensure that they are ready for release into a production environment.

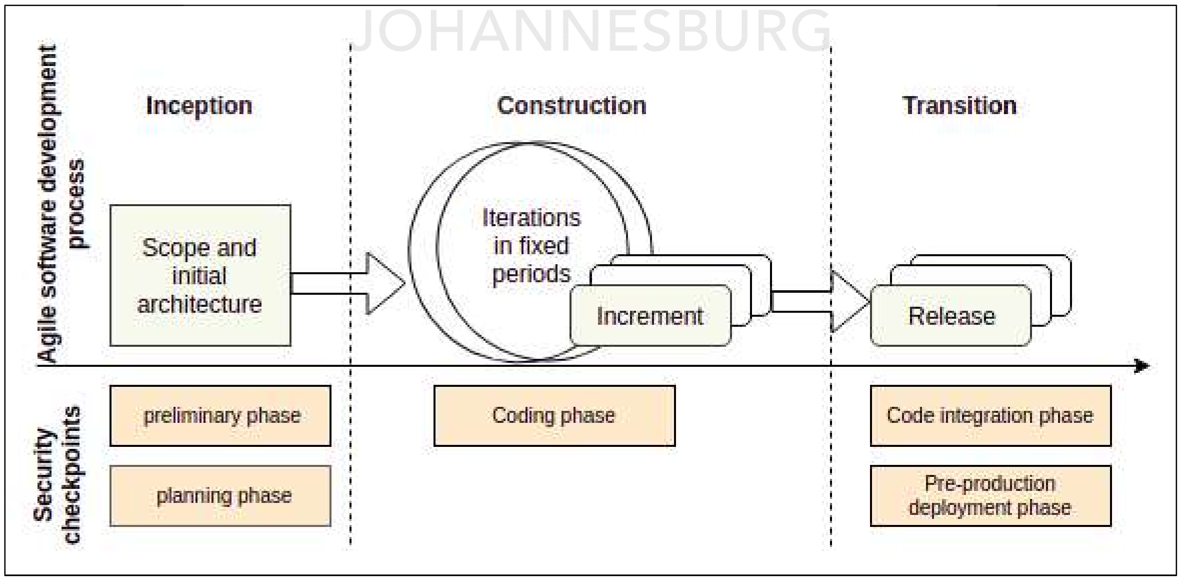


Figure 8.2. The relationship between SAFEMicroservices phases and Agile methodology stages

The sections below discuss how the SAFEMicroservices was used to develop PickMeUp in conjunction with the Agile software development methodology. The discussion is structured according to the three Agile methodology stages namely inception, construction and transition as shown in Figure 8.2 above.

The next section discusses the PickMeUp inception stage.

8.5 Inception stage

Following the Agile software methodology, the inception stage was used by the researcher to define the scope of PickMeUp and the initial model of the initial architecture using the functional requirements. In addition, the inception stage was also used to ensure the documentation of the security requirements of PickMeUp. Recall that documenting the security requirements of microservices composition (SDA-1) is one of the secure development activities identified in Chapter 6. In this regard, the Agile methodology's inception stage was used to perform a risk assessment to identify threats to and vulnerabilities of PickMeUp along with their associated impacts. The SAFEMicroservices security-oriented activities discussed in the framework's preliminary and planning phase assisted to reach the goal of documenting the security requirements of PickMeUp.

The discussion in each of the next section provides a list of SAFEMicroservices activities performed in the Agile methodology inception stage, and then discuss each activity. First, the use of the SAFEMicroservices preliminary phase activities in the development of PickMeUp is discussed below.

8.5.1 SAFEMicroservices preliminary phase activities

The use of SAFEMicroservices in the development of PickMeUp provided the researcher with an opportunity to extend the Agile software methodology's inception stage with four activities from SAFEMicroservices shown in Table 8.1 below. Table 8.1 provides a list of security activities prescribed by the preliminary phase of SAFEMicroservices and the deliverables of each activity.

SAFEMicroservices was a convenient framework in the sense that the preliminary phase not only provided guidance on how to perform the four activities listed in Table 8.3 but also provided ready-made reusable security artefacts that were used in the development of PickMeUp. This took away the burden of performing the cumbersome, time-consuming activities of the SAFEMicroservices preliminary phase from the beginning. This made it easier to gain a good understanding of the security risks associated with PickMeUp without spending much time performing an architecture-centric threat modeling, threat and vulnerability analysis.

Table 8.1. SAFEMicroservices preliminary phase activities

Preliminary phase activities	**Required security artefacts**
A.1 Architecture-centric threat modeling	A.1.1 Security threats and associated vulnerabilities
A.2 Threats and vulnerabilities root cause analysis	A.2.2 Microservices architecture common weaknesses Enumeration
A.3 Architecture-level secure coding guidelines identification and classification	A.3.1 Catalogue of architecture-level secure coding Guidelines
A.4 Creation of development infrastructure	A.4.2 Secure build pipeline

The application of four SAFEMicroservices activities of the preliminary phase in PickMeUp is discussed below.

(a) A.1 Architecture-centric threat modeling

The researcher performed the threat modeling of PickMeUp using the architecture-level security threats and their associated vulnerabilities security artefact shown as A.1.1 in Table 8.1. The SAFEMicroservices architecture-centric threat modeling approach discussed in Chapter 5 provided the list of the architecture-level security threats and their associated vulnerabilities. The security artifact provided an understanding of the security risks in PickMeUp. The artefact also provided the researcher with a foundation towards a systematic approach to make and evaluate design decisions for PickMeUp. Besides, the list assisted the researcher in identifying potential design flaws in PickMeUp that could not easily have been found using other techniques such as code reviews and static source analysis. For example, without an architecture-centric approach, a threat such as insecure microservices discovery and the associated vulnerabilities could not have been easily identified. In addition to the artefacts provided by SAFEMicroservices, a risk assessment was further conducted to identify risk in the basic functional requirements of PickMeUp that is not rooted in the architecture. The security threats were used as high-level *security goals* for PickMeUp. This is discussed further in the coding phase below.

(b) A.2 Threat and vulnerabilities root cause analysis

The researcher performed the root cause analysis using the microservices architecture common weakness enumeration security artefact, shown as A.2.2 in Table 8.1 and provided in Appendix A as A.2.2. The SAFEMicroservices threats and vulnerability root cause analysis method provided the artefact as a ready-made catalog of common vulnerabilities that apply to PickMeUp. As a result, there was no need to perform a detailed threat and vulnerability root cause analysis for PickMeUp. The catalog provided by SAFEMicroservices was a quick guide toward understanding the various architectural decisions that impact the security of each component of PickMeUp. For example, to limit the security threats of insecure microservices API, the researcher could quickly identify what architectural decisions are the root cause of the threat. This gave the researcher guidance on what to consider when designing the APIs on microservices in PickMeUp.

(c) A.3 Architecture-level secure coding guidelines identification and classification

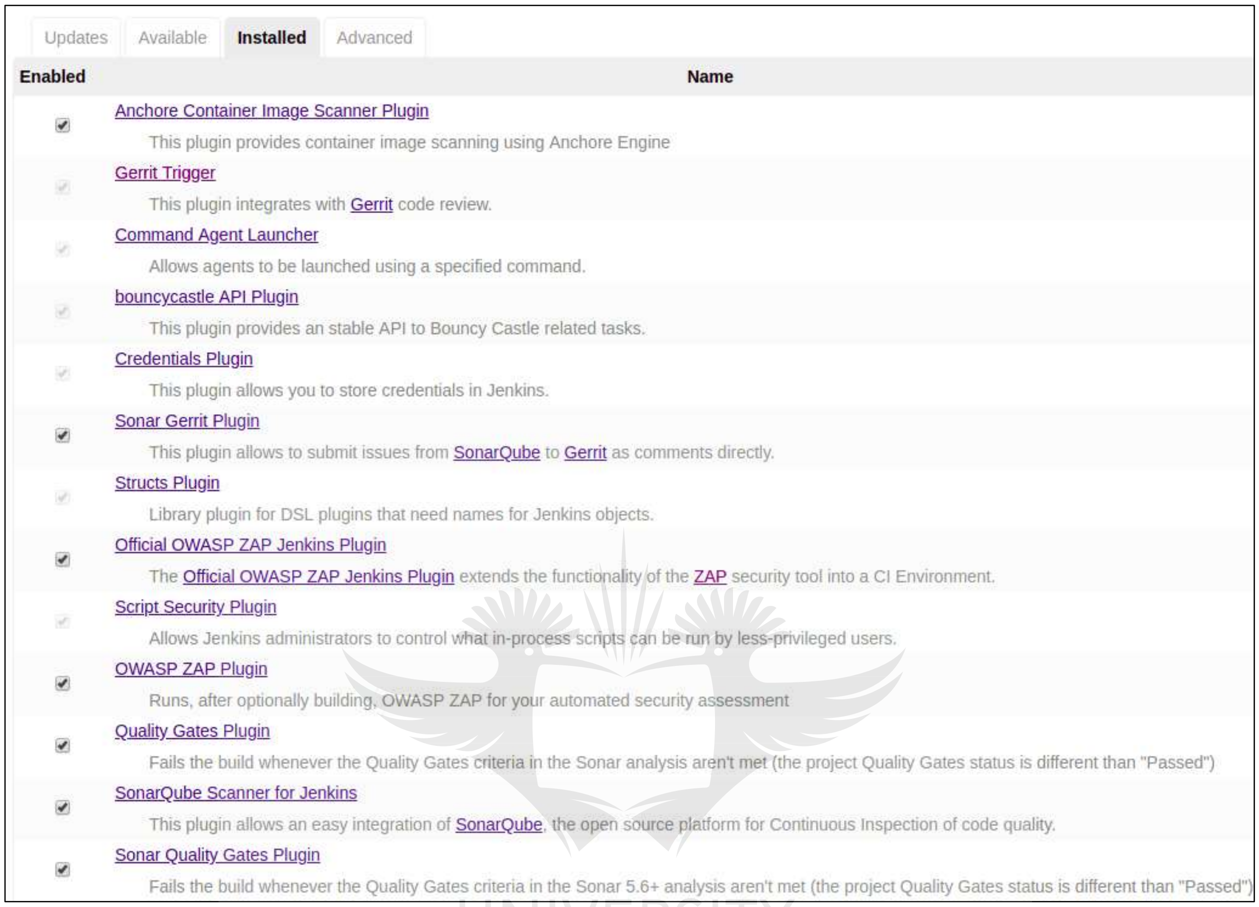
The researcher referred to the architecture-level secure coding guidelines identification and classification security artefact shown as A.3.1 in Table 8.1 and provided in Appendix A as A.3.1 to understand secure coding guidelines to apply in the development of PickMeUp. The SAFEMicroservices framework provided a list of architecture-level secure coding guidelines that provided the guidance required to design and create secure microservices design for PickMeUp from the ground up. The guidelines provided in Appendix A as A.3.1 augmented the language specific guidelines for the Java framework that was used to develop PickMeUp. The application of the secure coding guideline is discussed later in the construction stage of PickMeUp.

(d) A.4 Development infrastructure set up

The inception stage of PickMeUp was also used by the researcher to set up the infrastructure for the development of various components. As indicated above, SAFEMicroservices aim to be technology-agnostic. In this regard, the following freely available tools were used to set-up the development infrastructure in line with the high-level workflow diagram of the development infrastructure provided in SAFEMicroservices on Figure 7.5.

- IntelliJ (IntelliJ 2011) was used as the integrated development environment (IDE).
- FindSecurityBugs static analysis tool was installed on IntelliJ. FindSecurityBugs was reviewed in Chapter 6.
- Jenkins (Berg 2012) was installed as a code integration tool. Jenkins provides native code integration pipeline features. SonarQube plugin was installed on Jenkins for static analysis. The Anchore plug-in (Anchore 2018) was installed on Jenkins to validate container images. Tasks for automating the compiling and executing security tests cases were created on Jenkins pipeline. The OWASP dependency-check (Long 2015) plugin was installed to validate the third-party libraries.
- Gerrit (McIntosh et al. 2016), a manual code review tool was configured to manage the microservices source code review process.
- Atlassian Bitbucket (Atlassian 2019) was configured as a version control system.

Figure 8.3 below shows the security-related plug-ins installed on the Jenkins code integration tool.



Updates | Available | **Installed** | Advanced

Enabled	Name
✓	Anchore Container Image Scanner Plugin This plugin provides container image scanning using Anchore Engine
✓	Gerrit Trigger This plugin integrates with Gerrit code review.
✓	Command Agent Launcher Allows agents to be launched using a specified command.
✓	bouncycastle API Plugin This plugin provides an stable API to Bouncy Castle related tasks.
✓	Credentials Plugin This plugin allows you to store credentials in Jenkins.
✓	Sonar Gerrit Plugin This plugin allows to submit issues from SonarQube to Gerrit as comments directly.
✓	Structs Plugin Library plugin for DSL plugins that need names for Jenkins objects.
✓	Official OWASP ZAP Jenkins Plugin The Official OWASP ZAP Jenkins Plugin extends the functionality of the ZAP security tool into a CI Environment.
✓	Script Security Plugin Allows Jenkins administrators to control what in-process scripts can be run by less-privileged users.
✓	OWASP ZAP Plugin Runs, after optionally building, OWASP ZAP for your automated security assessment
✓	Quality Gates Plugin Fails the build whenever the Quality Gates criteria in the Sonar analysis aren't met (the project Quality Gates status is different than "Passed")
✓	SonarQube Scanner for Jenkins This plugin allows an easy integration of SonarQube, the open source platform for Continuous Inspection of code quality.
✓	Sonar Quality Gates Plugin Fails the build whenever the Quality Gates criteria in the Sonar 5.6+ analysis aren't met (the project Quality Gates status is different than "Passed")

Figure 8.3. Security plug-ins installed on Jenkins.

In order to ensure the security of the development infrastructure for PickMeUp, the following access controls measures were put in place as per recommendations in SAFEMicroservices:

- Access to Jenkins administration console was limited using a username and password.
- Access to BitBucket was limited to the use of secure socket shell (SSH) access keys.
- Access to Gerrit was also limited to the use of SSH access keys.

The next section discusses the use of SAFEMicroservices to plan for security during the inception stage of PickMeUp.

8.5.2 SAFEMicroservices planning phase activities

The use of SAFEMicroservices in the development of PickMeUp provided the researcher with an opportunity to extend the Agile software methodology inception stage with two SAFEMicroservices activities shown in Table 8.2 below. Table 8.2 provides a list of security activities prescribed by the planning phase of SAFEMicroservices and the deliverables of each activity. The SAFEMicroservices planning phase also provided the design artefacts that describe how security controls should be positioned to maintain confidentiality, integrity, availability, and non-repudiation in PickMeUp. As a result, the SAFEMicroservices manual, cumbersome and time-consuming activities of the planning phase were not required to be performed in detail by the researcher in the development of PickMeUp.

Table 8.2. Planning phase activities and deliverables

Planning phase activities	**Required security artefacts**
B.1 Microservices abuse or misuse cases identification	B.1.4 Microservices abuse cases & protection measures
B.2 Creation of security architecture	B.2.1 security standards B.2.2 Secure design principles B.2.3 Monitoring and adaptation mechanisms

The application of two SAFEMicroservices activities of the planning phase in the development of PickMeUp is discussed below.

(a) B.1 Microservices abuse or misuse cases identification

The security artefact depicted *B.1.4* on Table 8.2 and provided in Appendix A as B.1.4 was used by the researcher identify abuse or misuse cases in microservices. As part of the approach to elicit microservices abuse and misuse cases, the SAFEMicroservices identify attack patterns associated with microservices. The SAFEMicroservices attack patterns were applicable to PickMeUp and provided an understanding of how to protect PickMeUp from such attacks. Besides, the SAFEMicroservices framework provided a ready-made list of microservices abuse cases and protection measures. The list was not only useful to the identification of protection measures, but it assisted in identifying tools and techniques to use to mitigate the attacks.

The researcher used the abuse cases provided by SAFEMicroservices for two purposes; first, to create *security user stories* for Agile methodology. The user, in this case, is a malicious attacker. The *security user stories* created were then linked to the *security goal* which was created from the security threat associated with the abuse cases. Secondly, the researcher used the abuse cases to construct security test cases in the coding phase. The security test cases were created from the attack scenarios elicited from attack patterns. The attack scenarios were designed to test the written source code to ensure that the microservices software satisfied the security requirements. Figure 8.4 below shows the relationship between security goals, security use stories and development tasks created for PickMeUp.

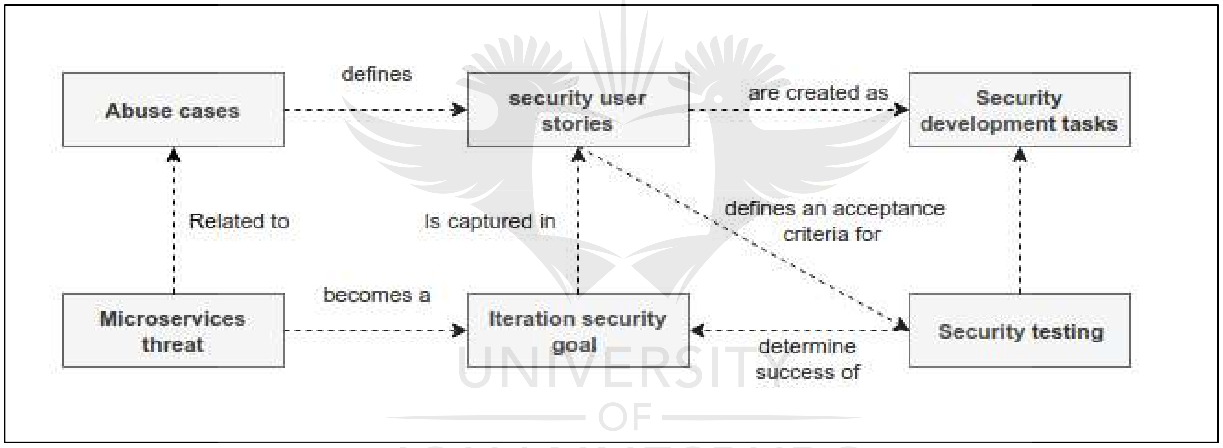


Figure 8.4. Conceptual model of relationship between abuse cases, user stories and goal

(b) B.2 The security architecture of a microservices composition

SAFEMicroservices provided the researcher with three important artefacts towards the creation of the security architecture of PickMeUp. The artefacts are *B.2.1 – B.2.3* on Table 8.2. First, the researcher adopted the security standards and secure design principles to ensure that the design of PickMeUp components were secure. These security standards also provided guidance towards creating secure designs and how to ensure that microservices in PickMeUp communicate securely. Secondly, the monitoring and adaptation mechanisms, shown as B.2.3 on Table 8.2, provided guidance on how to design microservices with monitoring and adaptation in mind. The design decisions are further discussed in the construction stage of PickMeUp.

8.5.3 Summary of the Inception stage

The adoption of SAFEMicroservices in PickMeUp provided the following general benefits in security requirements elicitation:

- SAFEMicroservices assisted the researcher in clarifying security requirements. Awareness of security was increased by an awareness of specific business assets that are at risk. The identification of PickMeUp security threats, and the abuse cases assisted in reasoning about security risk in concrete terms. This helped ensure clarity of security requirements in this research.
- SAFEMicroservices allowed security requirements not only to be derived from functional requirements as implicit requirements but also from the architecture and the technology. This assisted in ensuring this research appreciates the importance of a broader security strategy.
- SAFEMicroservices provided a reusable development infrastructure that allowed automated validation of security. Automated security testing provided timely feedback on detected security issues on the source code. This is discussed further in the next phases.

The next section discusses the construction stage of PickMeUp.

8.6 Construction stage

In the Agile methodology, the construction stage is for developing software in a set of iterations. For each iteration, the researcher determined the goal of the iteration and selected a set of functional user stories to achieve the goal. Functional requirements were elicited and captured as functional user stories, and the software code was written incrementally to address the requirement. At the end of the iteration, the artefact was potentially shippable. This section discusses the application of the SAFEMicroservices coding phase in the development of PickMeUp.

8.6.1 Coding phase

The use of SAFEMicroservices in the construction stage of the Agile software development methodology allowed the construction stage to be extended with four activities shown in Table 8.3. The functional user stories and the security user stories discussed above were used to guide the writing of microservices source code. At the beginning of each iteration, the functional features of PickMeUp to be developed were defined. The security goal of the iteration was provided by the security threat of the component of the PickMeUp that was under construction in the iteration as per Figure 8.5 above. The user stories linked to the security goal were added into the iteration to ensure that software increments were constructed with security in mind.

Table 8.3. Coding phase activities and deliverables

SAFEMicroservices activities	**Security artefacts produced**
C.1 Writing security tests for the PickMeUp	C.1.2 Suite of security tests cases
C.2 Designing and writing secure microservices and infrastructure code	None
C.3 Static analysis on the code and manual code review	None
C.4 Perform manual security code review	C.4.1 Runtime infrastructure template C.4.2 Microservices code
C.5 Fix failed security test cases	

The construction phase used the six reusable artefacts of SAFEMicroservices shown on Table 8.4 below. Details use of each artefact is provided in the discussion of each activity below.

Table 8.4. Artefacts required for coding phase activities and deliverables

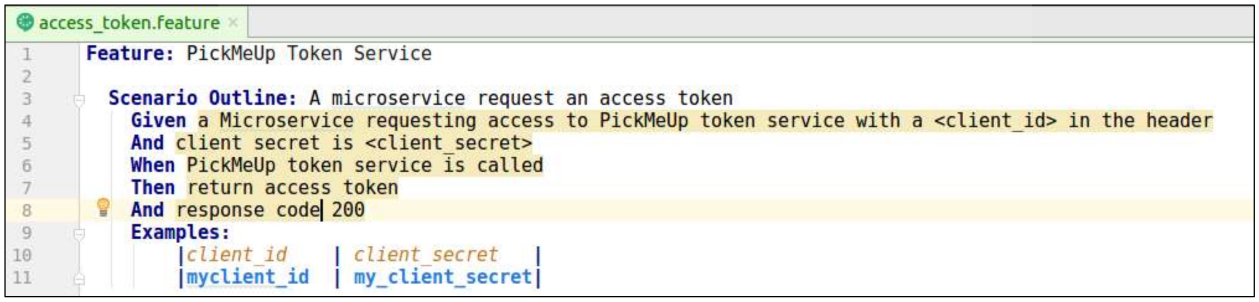
Coding phase required security artefacts
A.3.1 Catalog of architecture-level secure coding guidelines
A.4.2 Secure build pipeline
B.1.4 Microservices abuse cases and protection measures
B.2.2 Security standards
B.2.3 Secure design principles
B.2.4 Monitoring and adaptation mechanisms
B.1.1 Platform-specific coding guideline, in this case, Java secure coding guidelines

The use of SAFEMicroservices activities listed in Table 8.3 is now discussed next.

(a) C.1 Security test cases

The researcher wrote two types of security test cases namely security unit test and acceptance test. Security test cases were written in a manner that validated the microservices code against secure design principles and microservices security standards. Microservices abuse cases were used as attack scenarios to test the microservices to ensure that security requirements were satisfied. Spring Boot Test provided utilities to assist with unit testing (Reddy, 2017). Also, The Hamcrest library (Acetozi, 2017) was added to the spring framework to assist in testing.

The acceptance tests were documented using a security testing framework that uses Behaviour Driven Development (BBD) concepts to create security specifications that are executable as standard integration tests as part of the Jenkins build process. The security test specifications were documented using the language Cucumber (Ye, 2013), a language based on Gherkin domain-specific language (Härlin, 2016) which is simple and allows software engineers and testers to write complex tests while keeping the test comprehensible even to non-technical users. Figure 8.5 shows an example of an executable acceptance test case written using Cucumber. The acceptance tests cases were executed as *integration tests* on the Spring Boot Framework. Integration testing focuses on testing if the components work well together.


access_token.feature

```
Feature: PickMeUp Token Service

  Scenario Outline: A microservice request an access token
    Given a Microservice requesting access to PickMeUp token service with a <client_id> in the header
    And client secret is <client_secret>
    When PickMeUp token service is called
    Then return access token
    And response code 200
    Examples:
        |client_id    | client_secret    |
        |myclient_id  | my_client_secret|
```

Figure 8.5. Example of an acceptance test case

(b) C.2 Design and write secure code and infrastructure code

The microservices security standards and secure designs principles provided in SAFEMicroservices were used to make designs decisions for PickMeUp. The researcher considered ten design decisions shown of Table 8.5 to guide the development of PickMeUp. Limiting the number of standards to comply with was meant to make the scope of PickMeUp manageable for this research.

Table 8.5. PickMeUp security standards

	PickMeUp security standards
1	Any client communication in PickMeUp must be done via the PickMeUp API Gateway
2	Every microservices in PickMeUp must authenticate to the PickMeUp API gateway
3	Each microservices in PickMeUp must have a unique API key for calling another microservice
4	All API requests in PickMeUp must be logged to a centralized logging and monitoring system
5	All communication in the PickMeUp must use Transport Layer Security
6	All PickMeUp microservices must run in an approved application container technology
7	Deployment of microservices in PickMeUp must be automated
8	Data available to a microservice in PickMeUp must be limited what the microservices requires to function
9	Microservices in PickMeUp must only be able to access messaging channels that they require to function
10	Development of microservices must follow the secure coding guidelines provided

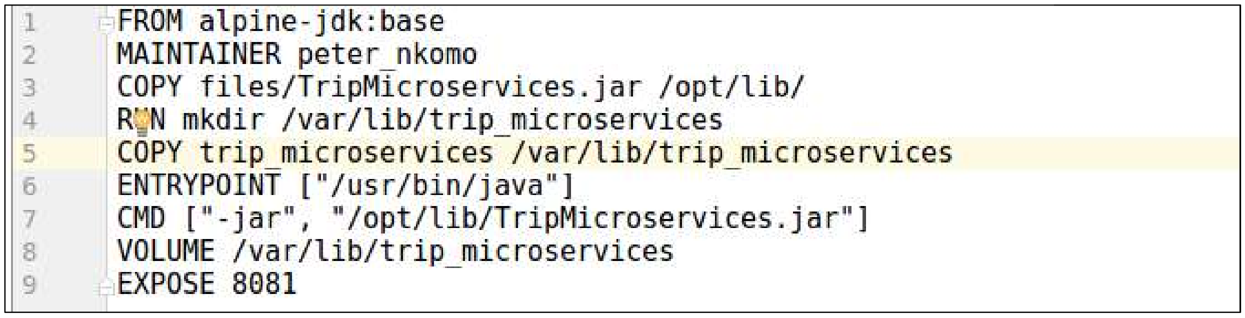
In order to ensure compliance with the standards in Table 8.5, the following additional technology choices were made to implement PickMeUp in addition to those discussed in section 8.2 above:

- Oauth2 (Guiterrez, 2016), an authorization framework was used for microservices authentication in PickMeUp. The framework seamlessly integrates into the Spring Boot framework using the Spring framework. Also, the JSON Web Token (Raman & Dewailly, 2018) was used to represent the claims secured between two communicating components in PickMeUp. JWT also seamlessly integrate into the Spring Boot framework.
- A microservices dedicated to monitoring was created for PickMeUp. The monitoring mechanisms on the microservice were built using the Hystrix (Christensen, 2012) and

Turbine (Netflix, 2018) libraries. Hystrix provides monitoring for all REST call and re-routing of a request in case of failure of a microservices. Turbine aggregated all Hystrix monitoring data into meaningful date for display on the dashboard. The functioning of the monitoring microservices is discussed later in the chapter.

- Each microservices implemented the circuit breaker pattern (Montesi &Weber, 2016) using the Netflix Hystrix (Montesi &Weber, 2016) library to ensure that each microservices were designed for failure. A circuit breaker pattern accepts microservices failures and tracks each failure by wrapping a call to a microservices a monitor. When a microservices is in the failed stage, circuit sends the error message without making a call to the microservices, and when the microservice is available, the request is sent to the microservice. Further details are discussed below in the monitoring section.

The development tasks of PickMeUp were represented as user stories and listed on the product backlog as mentioned. Development of the features of PickMeUp was done in iterations. An agile development board was created on the Atlassian Jira (Fisher, Koning &Ludwigsen, 2013) to keep track of the development task of PickMeUp. The microservices were written in Java programming language and packaged as Docker images. Figure 8.6 below shows an example of a script to package and deploy the trip management microservices as a docker image. The microservices requires port 8081 to function. Chef configuration management tool was used to automate the provision of Docker containers.

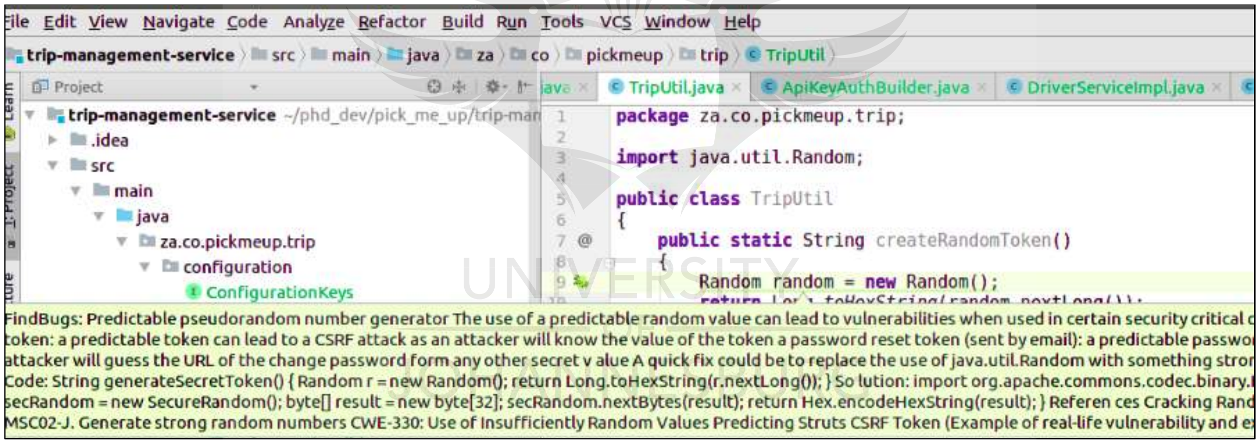


```
FROM alpine-jdk:base
MAINTAINER peter_nkomo
COPY files/TripMicroservices.jar /opt/lib/
RUN mkdir /var/lib/trip_microservices
COPY trip_microservices /var/lib/trip_microservices
ENTRYPOINT ["/usr/bin/java"]
CMD ["-jar", "/opt/lib/TripMicroservices.jar"]
VOLUME /var/lib/trip_microservices
EXPOSE 8081
```

Figure 8.6. Script to deploy trip management microservice as Docker image

(c) C.3 Static analysis and manual review

Microservices source code changes done by a researcher on the local machine were tested by executing security unit tests, and static analysis was done using the FindSecurityBug static analysis tools. Figure 8.7 below shows an example of the security issues identified by the static analysis tool on the researcher's IDE. In Figure 8.7 the static analysis tool was able to detect a weak random number generator that the researcher had used in the source code. The tool provided a suggestion to mitigate the issue. Performing security testing on the local machine allowed the researcher to quickly identify security issues on the source code and to quickly fix the issues. This ensured timely feedback on security issues to the researcher. After fixing security issues reported on the IDE, the code was submitted to the Gerrit system for manual review to help identify any violation of policies, standards or design flaws that could not be identified by the static analysis tools.


ile Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
trip-management-service › src › main › java › za › co › pickmeup › trip › TripUtil
Project
java TripUtil.java ApiKeyAuthBuilder.java DriverServiceImpl.java
trip-management-service ~/phd_dev/pick_me_up/trip-man
.idea
src
main
java
za.co.pickmeup.trip
configuration
ConfigurationKeys

```
package za.co.pickmeup.trip;

import java.util.Random;

public class TripUtil
{
    public static String createRandomToken()
    {
        Random random = new Random();
```

FindBugs: Predictable pseudorandom number generator The use of a predictable random value can lead to vulnerabilities when used in certain security critical c
token: a predictable token can lead to a CSRF attack as an attacker will know the value of the token a password reset token (sent by email): a predictable passwo
attacker will guess the URL of the change password form any other secret v alue A quick fix could be to replace the use of java.util.Random with something stron
Code: String generateSecretToken() { Random r = new Random(); return Long.toHexString(r.nextLong()); } So lution: import org.apache.commons.codec.binary.
secRandom = new SecureRandom(); byte[] result = new byte[32]; secRandom.nextBytes(result); return Hex.encodeHexString(result); } Referen ces Cracking Rand
MSC02-J. Generate strong random numbers CWE-330: Use of Insufficiently Random Values Predicting Struts CSRF Token (Example of real-life vulnerability and e

Figure 8.7. Static analysis on IntelliJ

8.6.2 Summary of construction phase

The use of SAFEMicroservices in the construction phase of PickMeUp provided the following benefits:

- SAFEMicroservices made it possible to get timely feedback on security issues on the microservices source code, and this made it possible to fix security issues early in the development process.

- SAFEMicroservices provided reusable microservices security artefacts that made it possible to focus more on writing microservices source code instead of performing security analysis.

8.7 Transition stage

As per the Agile software methodology, the transition stage was used by the researcher for integration testing and for hardening the increment of PickMeUp to make them ready as a release for use in a production environment. In Chapter 5, it was noted that secure development of microservices requires that software engineers adopt secure programming best practices (SDA-2), validate security requirements and secure coding guidelines (SDA-3) and also securely configure the runtime infrastructure (SDA-4). The transition stage of PickMeUp provided an opportunity to the researcher to integrate activities of the SAFEMicroservices' coding integration phase and pre-production deployment phase to assist in this regard. The discussion in this section first provides a list of SAFEMicroservices activities performed in the Agile methodology transition stage, and then the discussion of each activity is presented.

First, the application of the SAFEMicroservices code integration phase activities is discussed below.

8.7.1 Code integration phase

The coding phase commenced as soon as the manual review was completed, and no security flaws were identified. The researcher extended the transition stage of the Agile software development methodology with three activities from SAFEMicroservices shown in Table 8.6.

Table 8.6. SAFEMicroservices code integration phase activities

SAFEMicroservices code integration phase activities
D.1 Build microservices and execute security test cases
D.2 Validate licences, libraries and container images
D.3 Fix failed security test cases

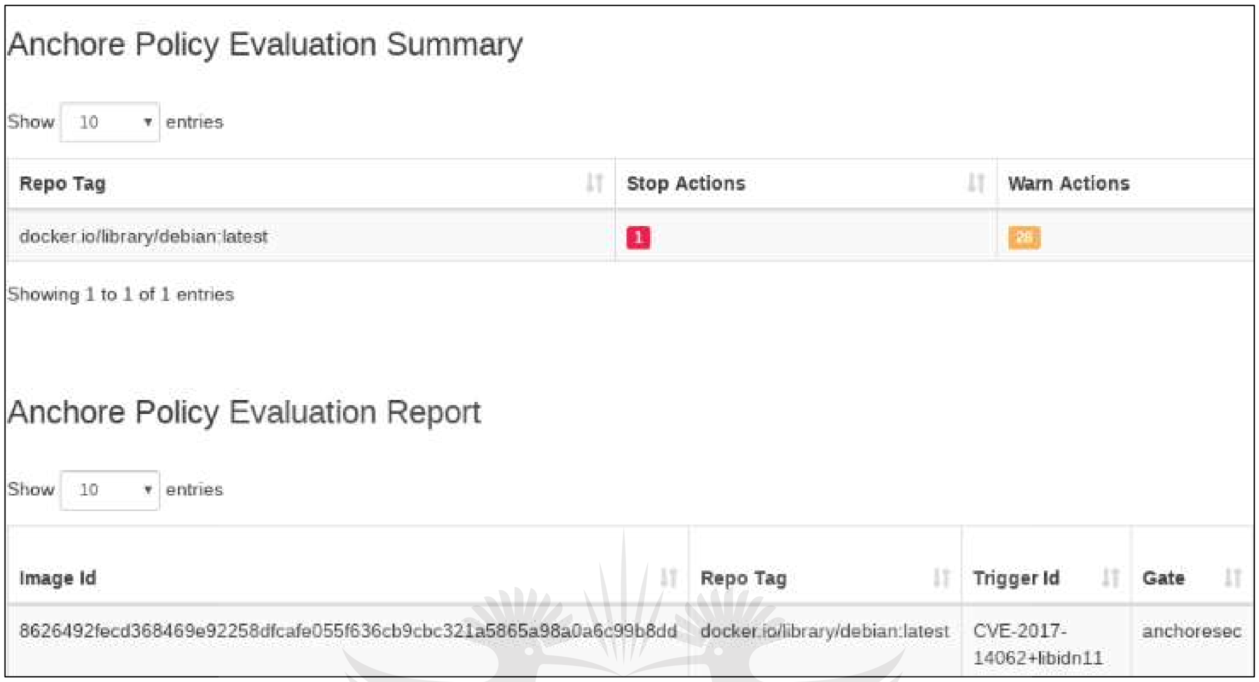
The use of each activity in PickMeUp is discussed below.

(a) D.1 Build microservices and execute security test cases

The Jenkins code integration tool compiled the source code of the microservices and then executed the security tests cases. The activity was automated.

(b) D.2 Validate license, libraries and container images

The Anchore plugin installed on Jenkins validated the docker images, and the OWASP dependency-check plugin validated the libraries to identify if microservices used trusted licenses. Figure 8.8 below shows an example of a dashboard provided by Jenkins to view the status of validation performed by the Anchore plugin.



Anchore Policy Evaluation Summary

Show 10 entries

Repo Tag	Stop Actions	Warn Actions
docker.io/library/debian:latest	1	28

Showing 1 to 1 of 1 entries

Anchore Policy Evaluation Report

Show 10 entries

Image Id	Repo Tag	Trigger Id	Gate
8626492fecd368469e92258dfcafe055f636cb9cbc321a5865a98a0a6c99b8dd	docker.io/library/debian:latest	CVE-2017-14062+libidn11	anchoresec

Fig 8.8. Anchore Docker image validation

(c) D.3 Fix failed security test cases

Any security issues that were reported during building of microservices and execution of security test cases or any failure during validation of licenses, libraries and container images were identified at this point. These security issues were added to the backlog as security user stories and the fixed in the subsequent iteration by the researcher. This activity aimed at ensuring that no microservices were deployed with security issues.

The next section discusses the deployment and operation of PickMeUp.

8.7.2 Pre-production deployment phase

The pre-production phase commenced as soon all the validations of the code integration phase were successful. The use of SAFEMicroservices in the transition stage of the Agile software development methodology allowed the transition stage to be extended with four activities shown in Table 8.7 below.

Table 8.7. Pre-production deployment activities

SAFEMicroservices pre-production deployment phase activities
E.1 Provision pre-production environment and deploy
E.2 Validate security requirements and infrastructure
E.3 Penetration testing
E.4 Fix failed security tests

The activities are discussed next.

(a) E.1 Provision preproduction deployment environment and deploy

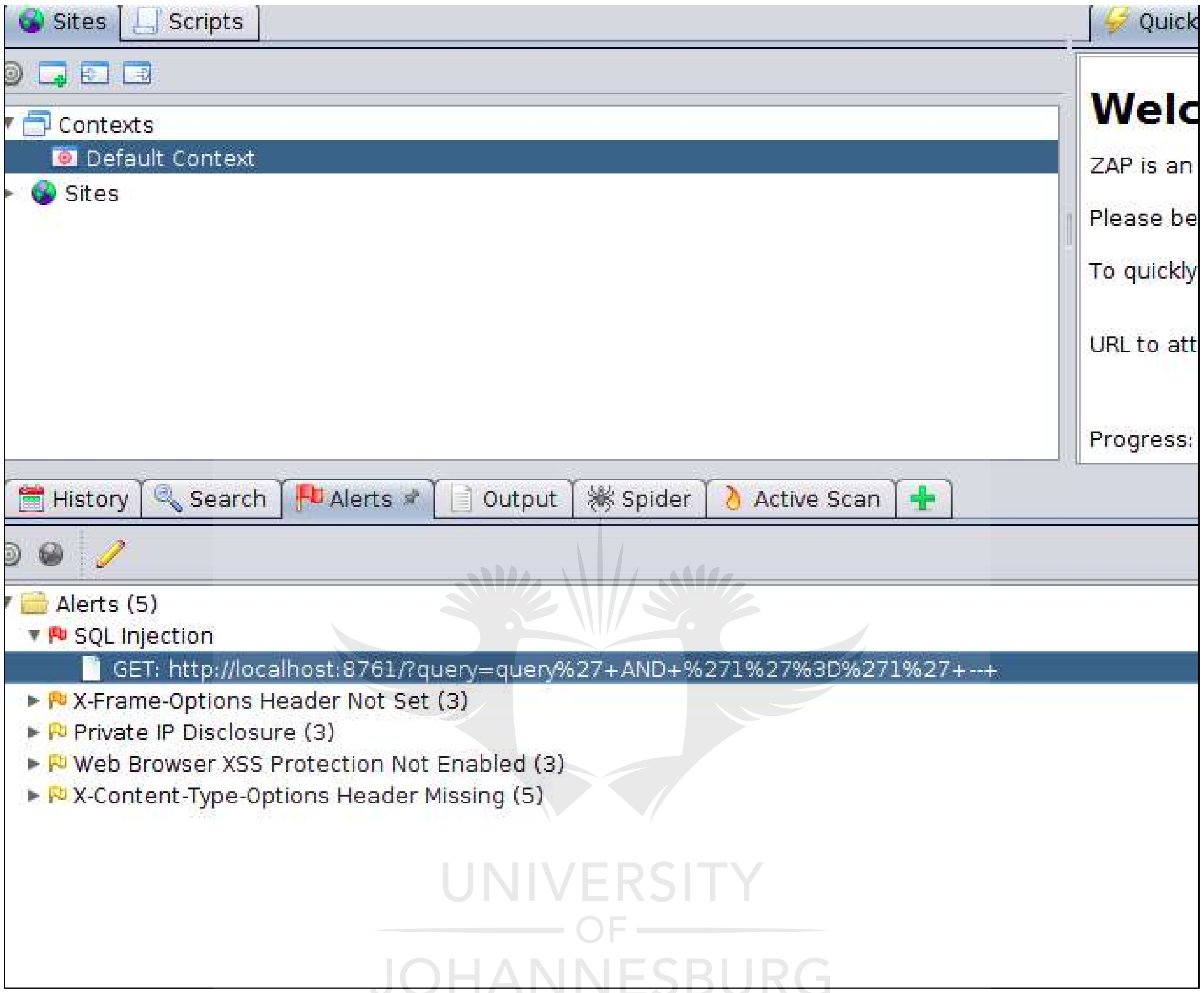
The provisioning of a pre-production environment was automated. The Chef configuration management tool was used to provision docker containers running each component of PickMeUp in conjunction with the infrastructure template, an artefact created and tested in the coding phase.

(b) E.2 Validate security requirement and infrastructure

The validation of security requirements was manual performed by the researcher to ensure authentication was being successfully done according to the security standards. In addition, the validation also made sure that microservices access to certain resources such as file and directory was only limited to what is necessary for the microservices to perform their function.

(c) E.3 Penetration testing

A basic penetration testing of PickMeUp was performed using the OWASP ZAP. Figure 8.9 below show an example of the OWASP ZAP plugin dashboard depicting security issues identified on the services registry that was developed for PickMeUp. Figure 8.9 shows a few security issues like SQL injection and cross site scripting identified by the tool. This tool can be deployed on a software engineer development machine and a scan performed before submitting the code to a common repository to quickly identify security issues.



Alerts (5)
SQL Injection
GET: http://localhost:8761/?query=query%27+AND+%271%27%3D%271%27+--+
X-Frame-Options Header Not Set (3)
Private IP Disclosure (3)
Web Browser XSS Protection Not Enabled (3)
X-Content-Type-Options Header Missing (5)

Fig 8.9. Microservices penetration testing

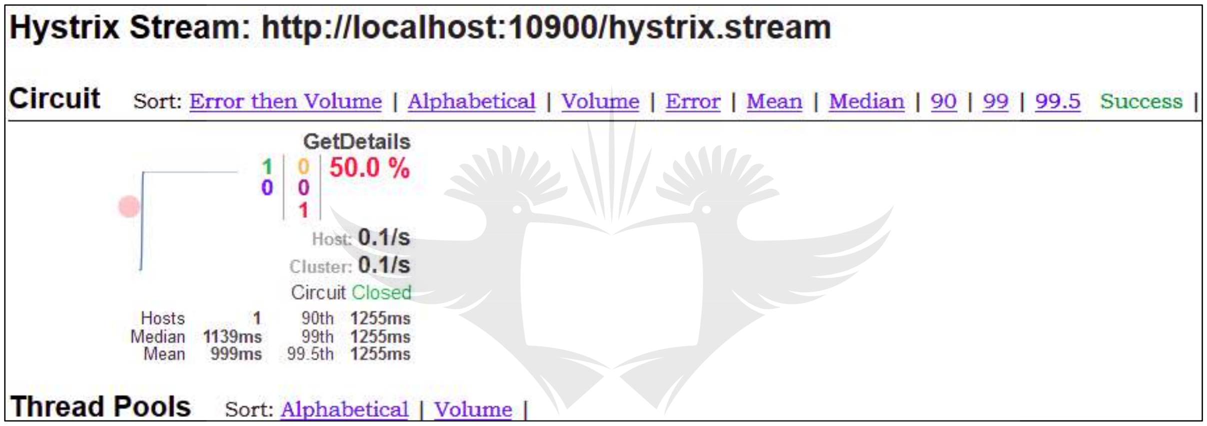
8.7.3 Summary of the transition phase

The use of SAFEMicroservices in the transition phase of PickMeUp enabled the integration of various security-focused tools into the build pipeline to ensure the automation of various security validation tasks. This simplified tasks such as security testing, and the provision of the runtime environment.

Finally, the integration of the SAFEMicroservices operational phase activities is discussed briefly.

8.8 The Operational phase

The essential activities in this phase were to ensure that instances of the microservices were monitored. Figure 8.10 below shows a simple dashboard showing the monitoring details of a microservices for getting passenger details. The circuit closed means that the microservices is currently available. However, the microservices was available fifty percent of the time so far. The Hystrix implementation in each microservices enabled monitoring microservices to gather information about the status of each microservice in PickMeUp.


Hystrix Stream: http://localhost:10900/hystrix.stream

Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5 Success |

GetDetails
1 | 0 | 50.0 %
0 | 0
1
Host: 0.1/s
Cluster: 0.1/s
Circuit Closed
Hosts 1 90th 1255ms
Median 1139ms 99th 1255ms
Mean 999ms 99.5th 1255ms

Thread Pools Sort: Alphabetical | Volume |

Fig 8.10. Microservices monitoring using Hystrix

8.9 Evaluation results

As mentioned above, the purpose of the implementation was to perform a basic empirical evaluation of SAFEMicroservices using three criteria. Below the evaluation based on the three-fold criteria is presented

- Easy to follow and effective steps – the activities of SAFEMicroservices can general be considered easy to follow. The researcher was able to follow the step of SAFEMicroservices and the tools integrated into the development environment were effective in identifying software weaknesses. However, the artefacts provided by SAFEMicroservices still requires background security knowledge to navigate and also understand. After the implementation of PickMeUp the view of the researcher is that it would be helpful to create an introductory document for SAFEMicroservices that can introduce security concepts to software engineers not trained in security.

- Easy support of security-focused tools – the tools can easily be integrated into the framework. For example, various testing tools were easily integrated into the continuous integration system as plugin. The installation is straight forward and easy to perform. Documentation on the tools is readily available on the web. The installation of tools is also a once-off activity and the researcher spend less than hour to install all the plugins on the Jenkins. However, the researcher still needed to learn how to use the tools. The observation of the researcher is that depending on the tool chosen, the use of tools may be less intuitive.
- Easy with which the framework could be used in an iterative software development method - the implementation of PickMeUp used the Agile methodology. The observations of the researcher are that although the implementation of PickMeUp can be considered a success, creating security test cases from abuse cases require more security knowledge that many software engineers may not possess. In that regards, there is still a need for someone in the team who possess good security knowledge to effectively implement SAFEMicroservices. In addition, there can be many abuse cases to consider for a simple microservice software change and a software engineer is expected to document all or most of the security test cases to test simple software change. This can be overwhelming and can affect the Agile principle of fast releases. There is therefore still a need to balance between the acceptable security level and the rate of software release when using SAFEMicroservices.

8.10 Conclusion

This chapter discussed the development of PickMeUp, an experimental microservices-based application using SAFEMicroservices. The purpose of the development of PickMeUp was to perform an empirical evaluation of SAFEMicroservices to determine through observation if the framework is adequately specified for the task of developing secure microservices. PickMeUp was developed from the ground up using the Agile methodology. The security-focused activities of SAFEMicroservices were used with success in the various phases on the development cycle of PickMeUp from inception to deployment. The various tools and techniques proposed in SAFEMicroservices were also used successful in the development of PickMeUp.

Although some of SAFEMicroservices phases such as the preliminary phase and the planning phase are mostly manual, time-consuming and cumbersome, SAFEMicroservices provided reusable security artefacts that simplified the development of PickMeUp for this research. The security threats and vulnerabilities catalog and the microservices architecture common weakness enumeration catalog provided the necessary foundation to perform a risk assessment of PickMeUp and to understand the risk of the application. The architecture-level secure coding guidelines, design principles and standards provided the necessary architectural knowledge required to make design decisions for PickMeUp that take security into consideration. In addition, the SAFEMicroservices artefacts empowered the researcher with the necessary information and protection measures to address the security challenges of PickMeUp. The use of various testing tools integrated into the development pipeline provided quick feedback on security vulnerabilities during the implementation phase of PickMeUp.

The implementation of SAFEMicroservices discussed in this chapter can be considered a success. The implementation discussed in this chapter also showed that SAFEMicroservices can be used in an iterative software development process.

In the next chapter, a theoretical evaluation of SAFEMicroservices is provided to augment the empirical evaluation performed in this chapter.

PART IV

Chapter 9

Evaluation of the SAFEMicroservices Framework

9.0 Introduction

In Chapter 7, a software framework for developing secure microservices called SAFEMicroservices was proposed. The aim of the framework is to provide guidance on how to develop secure microservices from the start by incorporating security-oriented activities into the microservices software development process. In Chapter 8, SAFEMicroservices was integrated with the Agile methodology, to develop an example microservices-based application called PickMeUp.

The aim of the implementation in Chapter 8 was not only to provide proof that the security-focused activities, tools and techniques proposed in SAFEMicroservices can be practically used to develop a secure application, but to also determine the suitability of SAFEMicroservices in an iterative and incremental software development process. Iterative and incremental software development processes generally provide a challenge when integrating secure software development practices (ben Othmane 2014). The implementation in Chapter 8 provided an empirical evaluation that applied the SAFEMicroservices in practice to establish if the theory outlined in the definitions of the framework can be successfully translated into a practical and meaning implementation (Shepperd & Ince 1993).

The next step is to perform a theoretical evaluation of the framework to establish if it is based on sound theory. As observed by Shepperd and Ince (1993), an evaluation of any model requires both a theoretical and an empirical evaluation. The chapter is organized as follows: Section 9.1 discuss the evaluation strategy. Section 9.2 discuss the evaluation of SAFEMicroservices. A conclusion of the evaluation is discussed in Section 9.3

9.1 Evaluation strategy

SAFEMicroservices can be defined as a process (IEEE 1990) since the framework defines a set of activities that can be used to develop, maintain and deliver secure microservices. In general, a good software process is one that delivers quality software and enhances software development productivity (Elsen, Liem & Akbar, 2016). This definition is based on the argument that an evaluation of a process model such as SAFEMicroservices is based on the hypothesis that security as a quality of microservices is determined by the quality of the activities or process used to develop the microservices. With this in mind, it becomes important in this evaluation to first defines security as a quality in the context of microservices. In this regard, a secure microservices can be defined as one that capture user input correctly, perform expected business functionality correctly and resist security breaches (Raghavan & Zhang, 2017). This implicates that it enforces data validation, functions as expected and secures its data (Raghavan & Zhang, 2017).

Since SAFEMicroservices is a quality process, using a quality model becomes a natural method to evaluate the framework since such a model encapsulate the concept of quality (ISO, 2011). The quality model is a set of characteristics and the relationships between characteristics which provide the basis for specifying quality requirements (ISO, 2011). The quality requirements in this context refer to security requirements. Also, common process models become vital to evaluate SAFEMicroservices as a software development process to determines its limitation and area of improvements (Noopur, 2006). This implies comparing SAFEMicroservices to existing process models that are designed for the same purpose (Noopur, 2006). With this in mind two approaches are chosen in this chapter to evaluate SAFEMicroservices namely:

i. By means of a quality model - this approach is referred to as the *SAFEMicroservices quality model* in this chapter

ii. By comparison with existing secure software development processes – this approach is referred to as the *secure software development processes comparison.*

Using the definition of security quality provided above, an evaluation question for SAFEMicroservices is formulated and provided below.

How adequately specified is SAFEMicroservices to provide security assurance that microservices developed using the framework are free from vulnerabilities, and that the microservices functions in an intended manner?

The two evaluation approaches listed above are now discussed.

9.2 SAFEMicroservices quality model

The evaluation of SAFEMicroservices discussed in this section adapts the quality model defined by ISO 25010 (ISO, 2011). The ISO 25010 quality model is chosen because it is the most recent model that defines a comprehensive list of quality characteristics. In addition, there is currently active and ongoing research work on using ISO 25010 to evaluate software development models. The work includes (Fontdevila et al,. 2017, König & Steffens, 2018, Estdale & Georgiadou, 2018) among many others.

The ISO 25010 quality model is now briefly defined as background.

9.2.1 ISO 25010 quality model

The ISO 25010 quality model provides a list of characteristics and sub characteristics that contribute towards quality. This research has identified the characteristics that apply to the evaluation of SAFEMicroservices. These characteristics are defined in Table 9.1 below in the context of this evaluation. The characteristics are used as evaluation requirements for SAFEMicroservices. They are used as a set of requirements that SAFEMicroservices should embody to be defined as an ideal process quality model.

Table 9.1. ISO 25010 quality model characteristics

Characteristics of quality model	Definition of characteristics	Sub-characteristic	Definition of sub-characteristics
Functionality suitability	Functional suitability means that the proposed framework fits the operational needs and requirements of developing secure microservices	Functional completeness	The extent to which the set of activities provided by the framework covers all the aspects of developing secure microservices.
		Functional appropriateness	The extent to which the framework facilitates the accomplishment of the goal of developing secure microservices.
Reliability		Reliability compliance	The extent to which the framework meets needs for reliability under normal operation.
Performance efficiency	Performance efficiency describes how the components of the frameworks execute efficiently.	Resource Utilization	The extent to which tools recommended in the framework meet acceptable levels when performing their functions.
Maintainability	Maintainability describes how easy it is to understand and adapt the various components of the framework to meet the needs of a development team	Modularity	The extent to which the framework advocate for discrete components such that a change to one component has minimal impact on the entire framework.
		Re-usability	The extent to which the artefacts of the framework can be re-used.
		Analysability	The extent of effectiveness and efficiency with which it is possible to understand and adapt the framework.
		Modifiability	The extent to which the framework can be extended by the software development team to meet specific needs.
		Testability	The extent of effectiveness and efficiency with which test criteria can be established for the framework.
Security	Security subsumes how to keep data safe	Confidentiality	The extent to which the framework ensures that data are accessible only to those authorized to have access.
		Integrity	The extent to the framework prevents unauthorized access to, or modification of information or assets.

Characteristics of quality model	Definition of characteristics	Sub-characteristic	Definition of sub-characteristics
		Non-repudiation	The extent to which actions or events in the framework can be proven to have taken place without any repudiation.
		Accountability	The extent to which the actions of an entity in the framework is traced uniquely to the entity.
		Authenticity	The extent to which the framework allows the identity of a subject or resource to be proved to be the one claimed.
Compatibility	Compatibility defines the ability of components of the framework to work together with other software products.	Co-existence	The extent to which a component of the framework performs their required functions efficiently in a shared environment without causing harm to other components running on the same environment.
		Interoperability	The extent to which components of the framework exchange information and use the information that has been exchanged.
Portability	Portability means that the necessary changes can be quickly done and easily installed.	Adaptability	The extent to which components of the framework can be adapted for different or evolving hardware, software or other operational or usage environments.
		Installability	The extent which component of the framework can be successfully installed and/or uninstalled in a specified environment.
		Replaceability	The extent to component of the framework can be replaced by another for the same purpose in the same environment.

The evaluation in this section takes the specification of SAFEMicroservices in Chapter 7 as input and assesses the specification against the characteristic provided in Table 9.1. The characteristic in Table 9.1 are used as a level of conformance to an ideal quality model. Next, Table 9.2 provides an assessment of SAFEMicroservices using the ISO 25010 characteristics and sub-characteristics.

Table 9.2. SAFEMicroservices evaluation using ISO 25010

Characteristics of Quality Model	**Sub-characteristic**	**SAFEMicroservices**
Functionality suitability	Functional completeness	• SAFEMicroservices defines activities, tools and method for developing secure microservices. The framework does not however define project management aspect of the secure development process. From a light-weight point of view the framework can be considered complete. In addition, SAFEMicroservices is designed in a manner that it can also be used with other secure development process were possible.
	Functional appropriateness	• SAFEMicroservices was used to develop a PickMeUp. The various artefacts were used with success to facilitate the accomplishment of the goal.
Performance efficiency	Resource Utilization	• The various tools that SAFEMicroservices recommend are being used in industry and are at an acceptable level when performing their functions.
Maintainability	Re-usability	• SAFEMicroservices provide the following reusable artefact, a catalog of security threats and vulnerabilities, architecture-level secure coding guidelines, secure design principles and security standards for microservices. These catalogs are generic and can be used by any team developing microservices.
	Analysability	• SAFEMicroservices guides software engineers to effectively incorporate the six secure development activities into the entire microservices development life cycle. SAFEMicroservices identify six critical phases in the software development process that are common to both sequential and iterative methodologies and apply to new trends in software development. These phases are used to integrate security-oriented activities, tools, and techniques into the development process. The framework is defined in such a way that it can be analysed.
	Modifiability	• SAFEMicroservices can be extended by a software development team to meet specific needs. The framework is generic in approach to security and allows engineers to use artefacts in their organization-specific environments
Security	Confidentiality Integrity Non-repudiation Accountability Authenticity	• SAFEMicroservices provide guidance on how to create a secure development environment and how to ensure the deployment infrastructure is secure to avoid the microservices build pipeline from being a weak link that attackers can use to breach security. Also, the protection measures provide the necessary guidance that engineers can use to develop secure microservices.

Characteristics of Quality Model	Sub-characteristic	SAFEMicroservices
Compatibility	Co-existence	• Various tools can be used in SAFEMicroservices. Software engineers can choose any security-focused tools that suit their environment. There is no expected detriment to other products deployed in the same environment.
Portability	Adaptability	• SAFEMicroservices is a systematic and flexible approach to security that accommodates variations in the implementation using different technologies and the risk profile of each microservice.
	Installability	• The tools used in SAFEMicroservices are easy to install. Software engineers can use the freely available plugin.
	Replaceability	• SAFEMicroservices is designed in such a way that the security-focused activities, techniques, and artefacts can be used with other frameworks.

Table 9.3 indicates that SAFEMicroservices conforms to most of the characteristic of ISO 25010 when evaluated using the eleven sub-characteristics. However, the following limitations of a theoretical evaluation are recognized in this research. The evaluation does not directly address the theory that underlies SAFEMicroservices but considers instead the theoretical basis expressed in the specification in Chapter 8 by the researcher. The evaluation is in actual fact measuring SAFEMicroservices indirectly using another quality process model, in this case ISO25010. The researcher recognizes that although conformance to the standards is a recognized way to evaluate a process model like SAFEMicroservices, as is the case when a process model requires international certification, validation using such an approach cannot be guaranteed to address the totality of the level of security expected from a framework to present an argument that SAFEMicroservices is a suitable framework. In addition, there is a need to test the requirements of ISO 25010 in the industry context to determine the consistence of the theory behind SAFEMicroservices to reality although this may be a costly exercise.

The next section compares SAFEMicroservices with other secure software development process.

9.3 Secure software development process model comparison

Although SAFEMicroservices focuses on developing secure microservices compositions, this section evaluates SAFEMicroservices by comparing the framework with existing secure software development methodologies that can be used for the same purpose. Three popular secure software development methodologies namely Microsoft's Security Development Life cycle (SDL) (Howard & Lipner 2006), OWASP's Comprehensive, Lightweight Application Security Process (CLASP) (OWASP 2006) and McGraw' Touchpoints (McGraw (2006)) are considered. These methodologies are recognized as significant players in the field.

A separate comparison of Microsoft's SDL, CLASP, and Touchpoints is provided in De Win et al. 2009. For the purposes of this thesis, the comparison by De Win et al. 2009 was revisited by reviewing the latest documentation of these methodologies to determine any process improvement that might have occur since the initial comparison. The CLASP book version 1.2 available from the OWASP website (OWASP, 2018) was used for CLASP. The book by Howard and Lipner (2006) was used for Microsoft SDL and the Microsoft web site was also checked for updates on the SDL. The book by McGraw (2006) was used to understand Touchpoints.

The comparison of the secure software development processes in this chapter is based on two areas namely:

- *Risk assessment process* – the activities of identifying threats and vulnerabilities
- *Software construction process* – the activities used to develop, test and deploy an application

The comparison in this thesis does not cover the initial project management aspects and security training that other secure software development processes consider important. This is already an accepted weakness of SAFEMicroservices.

The next section discusses the comparison.

9.3.1 Risk assessment process comparison

The risk process is about identifying threats to and vulnerabilities of a given system. The comparison discussed in this section is based on the specification of threat modeling and how elicited threats and vulnerabilities are analyzed in each secure software development process. The comparison is meant to provide a high-level view and does not consider the depth of the specified methodology for each category. Table 9.3 document the comparison using a few sets of basic security-focused activities. A tick (√) means that the framework provides security-focused activities or guidance that address the requirements and an *x* means that there is no guidance or specification of the activities.

Table 9.3 Risk assessment process comparison of SAFEMicroservices to other frameworks

Risk Assessment		**Microsoft SDL**	**CLASP**	**Touch-points**	**SAFEMicroservices**
1. Architecture-level threat modeling	1.1 Develop system understanding	√	√	√	√
	1.2 Identify the external dependency	√	x	x	√
	1.3 Identify threats and threat types	√	√	√	√
	1.4 Assign risk to threats	√	√	√	√
	1.5 Perform weakness analysis	x	x	√	√
	1.6 Identify countermeasures	√	√	x	√
2. Analysis-level threat modeling	2.1 Perform vulnerabilities analysis	x	x	√	√
	2.2 Identify and describe threats agents	x	√	√	x
	2.3 Elicit and describe abuse cases	x	√	√	√
	2.4 Attack-pattern elicitation	x	x	√	√
	2.5 Rank misuse case	x	x	√	x
	2.6 Identify protection measures	x	√	x	√
	2.7 Document security architecture	x	√	x	√

Table 9.3 above shows that among the few selected security activities in the risk assessment process, of the three popular frameworks TouchPoint covers most areas of the architecture-level

threat modeling and analysis-level threat modeling. Microsoft's SDLC and OWASP generally do not adequately specify how to perform an attack-driven analysis and elicit abuse cases. On the other hand, SAFEMicroservices provides guidance on most of the security-focused activities in the risk assessment process but does not identify and describe threats agents and does not assign a ranking to misuse cases. Risk rating, however, is generally an arbitrary exercise (De Win et al. 2009). Besides, Table 9.3 shows that from a high-level view SAFEMicroservices does have a broader scope on both the architecture-level and analysis-level compared to the existing framework. This research does acknowledge the limitation of SAFEMicroservices that it is specified for a microservices architectural style compared to other frameworks that are architectural agnostic. In addition, not evaluated in this thesis is the depth in which each of the areas listed in Table 9.3 is covered in SAFEMicroservices.

The next question that should be asked in the evaluation of SAFEMicroservices is whether the security-focused activities and resulting artefacts that are produced in the risk assessment process are fit for software engineers. To answer this question, a threat to the validity of both the architecture-level threat modeling and the threat analysis is identified. First, the architecture-level threat modeling process in SAFEMicroservices leverages the Microsoft SDL threat process which is a well-tested process (Win et al.2009). The threat classification approach uses the Microsoft STRIDE, again a tested methodology. Although SAFEMicroservices does not assign a rating on the misuse cases, a rating is provided on the likelihood of an attack which is derived from the attack pattern in the CAPEC dictionary, a community-driven database of common attacks. Also, the analysis of the security threats and software weaknesses leverages a community-driven database of common weakness and common attack patterns. This makes the SAFEMicroservices threat modeling and analysis fit for software engineers, barring errors that software engineers can make in both architecture-level threat modeling and analysis.

The next section compares the construction process of the popular secure software development frameworks with SAFEMicroservices.

9.3.2 Software construction process comparison

The comparison in this section considers the activities provided by each secure development process for creating secure development infrastructure, coding, testing, deployments and support. Figure 9.4 below shows a comparative view of the development processes.

Table 9.4 Construction process comparison of SAFEMicroservices with other frameworks

Implementation		**Microsoft SDL**	**CLASP**	**Touch-points**	**SAFEMicroservices**
1. Secure development infrastructure	1.1 Secure use of version control system	x	x	x	√
	1.2 Use of branching strategy	x	x	x	√
	1.3 Secure use of code integration tool	x	x	x	√
	1.4 Support for automated security testing	x	x	x	√
2 Coding	2.1 Use of security standards	x	√	x	√
	2.2 Use of secure coding guideline	√	√	x	√
	2.3 Address reported issues	x	√	x	√
	2.4 Validate remediation	x	√	x	√
3 Testing	3.1 Build test cases using risks	x	x	√	√
	3.2 Execute security tests	x	√	x	√
	3.3 Perform unit testing	x	x	√	√
	3.4 Perform integration testing	x	x	√	√
	3.5 Static code analysis	x	√	x	√
	3.6 Manual code reviews	√	x	x	√
	3.7 Penetration testing	√	x	√	√
	3.8 Validation of external libraries	x	√	x	√
	3.9 Validate container or infrastructure	√	x	x	√
	3.10 Perform risk-based security testing	x	x	√	√
	3.11 Validate correct use of tools	√	x	x	x
	3.11 Security report	√	x	x	√

Implementation	Microsoft SDL	CLASP	Touch-points	SAFEMicroservices	Implementation
4 Deployments and Support	4.1 Code Sign-off	√	x	x	x
	4.2 Configure monitoring and logging	x	x	√	√
	4.3 Safe runtime configuration	x	x	x	√
	4.4 Verify infrastructure	x	x	x	√
	4.5Secure deployments	x	√	x	√
	4.6 Vulnerabilities reporting	√	√	x	√
	4.7 Fix security issues	√	√	x	√

Figure 9.4 above shows that among the few selected security-focused activities common to the construction process, existing secure software development processes do not provide guidance on how to set up a secure development environment. Very little is mentioned concerning such important aspect of software development such as the security of the code repository, continuous integration as well as source code branching and source code merging policies. SAFEMicroservices provide guidelines on how to create a development infrastructure, how to ensure comprehensive testing and deployments. Table 9.4 shows that on a high-level, SAFEMicroservices provides comprehensive end-to-end guidelines on how to ensure the development of secure microservices.

Although SAFEMicroservices does provide guidance to software engineers in the entire software development life cycle, the researcher does recognize that SAFEMicroservices is a light-weight framework and does not go into much details when discussing security-focused activities compared to other secure development frameworks. Framework such as Microsoft SDL and Touchpoints have rich literature in the form of books and websites written specifically for those frameworks and are tried and tested in the field. With this in mind, the view of the researcher is that SAFEMicrosevices can augment existing frameworks rather than be a competitor. In addition to guidance provided by existing frameworks, SAFEMicroservices can be used to address areas where existing frameworks are limited.

9.4 Conclusion

A theoretical evaluation by comparison of SAFEMicroservices was discussed in this chapter to augment the empirical evaluation done in Chapter 9. The evaluation in this chapter was based on two approaches, the use of a quality model and by comparing SAFEMicroservices with other software development processes that are similar in nature. The limitation of each of the approach was acknowledged in the discussion.

Taking into consideration the limitations of the evaluation approach and the identified limitation of SAFEMicroservices, the framework is adequately specified to provide grounds for confidence that microservices compositions developed using the framework are free from vulnerabilities. The researcher acknowledges however that although a framework is important towards the development of secure software, other factors not considered in this research may impact on the quality of software. The view supported by the research is that by using SAFEMicroservices, software engineers can perform the security-focused activities in any software development methodology, taking into consideration the organisation's culture and the technology landscape, thereby ensuring that the chances of successful adoption of the framework is enhanced. The limitation of SAFEMicroservices is that it is specified for the microservices architectural style and need to be extended to be useful in team using other software architectures.

Chapter 10

Conclusion and Future Work

10 Introduction

This thesis proposed a software development framework for secure microservices. The primary objective of the proposed framework is to guide software engineers to develop microservices from the ground up so that such microservices are inherently secure. The framework seeks to wean software engineers from a reactive approach to security where software security receive attention as and when security breaches occur. Reactive approach to security is expensive because it requires re-engineering efforts that are often required after microservices are written or a security breach has occurred.

This thesis introduced the microservices architectural style and its security challenges. A risk assessment of the architectural style was performed to identify the security threats and vulnerabilities in the architecture. The identification of the risks allowed for the identification of security-focused activities that are required to be performed by software engineers in their day-to-day development tasks. The activities referred to as secure development activities in this thesis are the foundation on which the proposed software development framework is built. Furthermore, the secure development activities were used to identify security-oriented tools and techniques that can assist in the day-to-day development tasks of secure microservices.

In Chapter 1 the motivation of this research was proposed together with the objective of the research. This chapter revisits the objectives to determine the success of the research. In addition, the research contributions are stated, and future research discussed. The chapter is then concluded

10.1 Revisiting the research objectives

The primary objective of this research is to propose a software development framework for secure microservices. The framework is constructed using a three main research questions. Secondary questions are proposed to assist in understanding the main research questions. In order to determine if the research objective has been met, the research questions defined in Chapter 1 are now revisited.

RQ1 - What are the security challenges associated with the microservices architectural style?

The state-of-the-art discussed in Chapter 3 identified the following five new security challenges of microservices:

i. Increased attack surface - an instance of a microservice is a unique network endpoint that requires a dedicated open network port to expose an application programming interface. Every instance of microservice require its own open port for communication. This gives an attacker an increased attack surface as new microservices are deployed across the network and an attack can be made on each microservices.

ii. Indefinable security perimeters - the deployment of microservices on containers result in containers being quickly set up anywhere within the network without any consideration for the traditional notion of demilitarized security perimeters.

iii. Security monitoring is complex – containers on a host machine can use network address translation which makes it challenging to identify network traffic coming to and from containers.

iv. Authentication is centralized - microservices deployed on separate containers presents a challenge of authenticating users and sharing user credentials between microservices in a symmetric and secure manner.

v. Threat modeling and risk assessment are localized - the microservices ownership model discussed in Chapter 3 emphasizes team autonomy and ensuring that threat modeling, and risk assessment is done before new versions of microservices are released becomes a challenge.

The following secondary research questions related to **RQ1** are addressed below:

(a) How does the microservices architectural style differ from common SOA implementations?

The state-of-the-art discussed in Chapter 2 and Chapter 3 of this thesis identified the following four key differences between microservices and traditional SOA implementations:

i. Services granularity – microservices are fine-grained components that focus on a single purpose and aim to do it well, whereas in SOA, a service can encapsulate a large product or a legacy system and is therefore course-grained.

ii. Security – traditional SOA implementations uses the enterprise service bus (ESB) as a security layer. Each service in SOA does not have to implement its security. On the contrary, in the microservices architecture, each microservices is an independent unit that is responsible for its security.

iii. Component sharing – traditional SOA implementations are based on making component reusable and shareable. In the microservices architectural style, each microservices is a single unit and is designed to have its data with minimal dependencies to ensure its autonomy.

iv. Service communication – a microservices communicate predominantly using a known application programming interface layer, whereas traditional SOA implementation uses a messaging middleware component responsible for mediation, routing, message enhancement, and protocol transformation.

(b) What are the security risks of microservices?

The risk assessment of the microservices architectural performed in Chapter 5 in this thesis identified the following five risks associated with microservices:

i. Insecure application programming interfaces - a weak set of application programming interfaces (APIs) exposes microservices to injection types of attacks, API manipulation and functionality misuse among other attacks.

ii. Unauthorized access - when there is no proper scalable identity access management system, a microservices is vulnerable to unauthorized access leading to tampering with data and information disclosure.
iii. Insecure microservices discovery - when microservices use discovery mechanisms that are not secure, spoofing, information disclosure and denial of service may occur.
iv. Insecure runtime infrastructure - containers and virtual machine, where microservices are deployed, may be compromised by the presence of errors or malware and an attacker can exploit the weakness to gain access to the microservices.
v. Insecure message broker - when the message broker is not correctly secured, spoofing, tampering with data, information disclosure and denial of service may occur.

(c) What methods can an attacker use to exploit weaknesses in microservices?

In Chapter 5, an analysis of security threats and vulnerabilities associated with microservices identified the following attack methods that a malicious agent can use when there are no sufficient protection measures:

i. Injection of unexpected items – an attacker can exploit the weaknesses on the microservices API validation mechanisms by manipulating the content that is sent as part of the request parameter.
ii. Use deceptive interactions – an attacker can deceive the microservices during an interaction in such a manner that the microservices allows the user to perform actions that they are not authorized to do.
iii. Abuse microservices functionality – an attacker can flood the microservices with many requests so that the microservices deplete its allocated computing resource while attempting to process the request. In so doing, the microservices may fail to provide functionality to legitimate users.
iv. Subvert microservices access control – an attacker can bypass authentication or authorization mechanisms to access the resources of a microservice illegally.
v. Use probabilistic techniques – an attacker can use brute force or send randomly created input data to microservices to analyze their failures and to discover certain assumptions made during the design of microservices.

vi. Collect and analyze information – an attacker can take advantage of insecure communication channels between communication microservices or gain access to microservices logs and read sensitive information.

The second main research question is now revisited below

RQ2 - How can software engineers build microservices in a systematic way in which security is an integral part of the entire microservices lifecycle?

In Chapter 5, as a starting point to address this question, the threat modeling exercise identified six important main activities to incorporate in the day-to-day software development tasks. The six identified activities are:

i. Document security requirements of microservices compositions
ii. Adopt secure programming best practices
iii. Validate security requirements and secure programming best practices
iv. Secure configuration of runtime infrastructure
v. Continuously monitor the behavior of components of the microservices composition
vi. Securely respond to attacks using adaptation mechanisms

Then in Chapter 7 this thesis proposed a software development framework that defined secondary security-focused activities to support the six activities above. The proposed framework provides a road map that can be used to ensure that the six activities are part of the daily software development tasks. The proposed framework defines a systematic way of integrating security in a software development process.

The secondary research questions related to **RQ2** are now addressed below:

a) **What are the design flaws associated with the microservices architectural style and how can they be avoided?**

The framework proposed in this thesis discussed an approach to analyse microservices threats and vulnerability that are rooted in the architectural style. The framework identified the following architectural design flaws:

i. Failure to comprehensively validate microservices inputs
ii. Failure to ensure data integrity of messages, deployment files and configuration files
iii. Failure to correctly identify, authenticate and authorize users
iv. Failure to limit access to microservices resources accordingly
v. Failure to limit the attack surface

b) **What guidelines can software engineers use to design and implement secure microservices and how can these guidelines be presented in a manner that is useful and convenient for software engineers especial those not trained in software security?**

The framework proposed in this thesis identified and constructed three important guidelines to address this research questions namely:

i. Secure coding guidelines – the secure coding guidelines in the thesis are elicited by performing a root cause analysis of security threats and vulnerabilities of microservices composition. The guidelines are architecture-centric to assist software engineers avoid design choices that lead to security flaws in microservices.
ii. Secure design principles – the principles were identified from the review performed in Chapter 6 and from a root cause analysis performed in the proposed framework. The design principles provide a set of rules that software engineers can use to avoid introducing vulnerabilities during the design of microservices.
iii. Microservices standards – the purpose of standards is to establish a set of mandatory requirements that software engineers must comply with in their daily development activities. The standards were created from a set of common security weaknesses and best practices suggested in literature.

c) **How can security-focused tools, techniques, and practices be integration in the development lifecycle to so that they become part of the software engineer's daily software development tasks?**

The software development framework proposed in Chapter 7 used the six main security-focused activities listed in research question RQ2 above in this chapter to identify security-focused tools, techniques and practices. The proposed framework then identified phases in the software development process where the six main activities are to be performed. This approach provides a method to appropriately determine how to integrate the tools, techniques and practices so that they become part of the software engineer's daily development tasks.

RQ3 - How can protection measures be correctly implemented and preserved to ensure that microservices are safe at all times?

The software development framework proposed in this thesis discussed the comprehensive use of guidelines to assist software engineers in implementing protection measures. The guidelines are discussed in the secondary question above. In addition, the proposed framework discussed security validation techniques that software engineers should use to ensure that protection measures and guidelines are correctly followed. Microservices should undergo extensive security testing before any deployment. The testing techniques defined in the proposed framework include:

i. Static security testing
ii. Manual code review
iii. Security unit test and acceptance test execution
iv. Penetration testing

10.2 Research contributions

This thesis provides a holistic security perspective of the microservices architectural style. First, the thesis identifies the security challenges of the architectural style using risk assessment techniques. The assessment brings to the fore the various security threats and vulnerabilities in microservices that are rooted in the architectural style. In addition, protection measures are suggested. The thesis also identified security-focused activities that should be incorporated in the development process of microservices. The tools are also reviewed to assist software engineers make an informed decision when choosing tools to incorporate into their daily development tasks. The security-focused activities are further used to identify tools, techniques and methods that can be used to improve the security of microservices.

This thesis also designs a catalog of microservices security threats, security weaknesses, and their mitigations. Software engineers can use the catalog as quick reference in their day-to-day microservices development tasks or as a manual to gain foundational knowledge to perform a risk assessment in a microservices-based software project.

Furthermore, the thesis designs a dictionary of coding guidelines to mitigate common microservices security weakness and common attacks on microservices. The dictionary is provided as a reusable artefact in a manner that is easy to use for software engineers who are not trained in security. Software engineers can use the dictionary as quick reference in their day-to-day microservices development tasks.

The thesis proposes a software development framework to build secure microservices from the ground up based on the identification of security-focused activities that are required in a microservices development lifecycle. The framework is specified in a manner that makes it agnostic to both culture and technology characteristics in a software development team to allow software engineers to apply software security controls within their unique organizational circumstances.

This research also proposed a secure development framework for secure microservices based on the assumption that software engineers are willing to adopt secure software development practices. The framework has defined best practices to follow in developing secure microservices but not much has been considered about the context in which software development team operate. Success of any software development approach is based on the ingenuity of various stakeholders and their interactions with one another, and this has not been considered or tested in this research. There is therefore a possibility that adopting the proposed framework may be considered out of touch with a team's culture, team skill set and team size. Furthermore, this thesis has also not identified the minimum security skill set required to effectively adopt the framework or proposed measures to deal with resistance of software engineers in a team in adopting some or all of the security-focused activities or use of tools.

10.3 Future work

One of the fundamental premises on which the proposed software development framework for secure microservices is based on, is that few software engineers are trained in security. To remedy the lack of security expertise, the framework provides ready-made catalogs of threats and protection measures, a catalog of secure design principles, and a set of security standards. Future work is to ascertain the effect of these catalogs on the development of security skills among software engineers. Besides, an investigation is required to determine whether the catalogs created in this thesis have sufficient information to be used as a security teaching aid. A qualitative analysis is required to provide guidance on which section of the catalog engineers both with security knowledge and those without consider vital.

Furthermore, there is need to investigate further how the framework can be effectively used in different software development teams as discussed.

10.4 Conclusion

This chapter has presented the conclusion of this thesis by revisiting the research objectives. The limitation of this research and the future research direction are also discussed. The main goal of this thesis was to propose a software development framework for secure microservices.

The framework proposed in this thesis first identify the risk associated with the microservices architectural style as a basis towards understanding the security challenges of microservices. The security threats and vulnerabilities identified in the risk assessment are then used to identify important security-oriented activities that should be incorporated into the daily development tasks. The identification of security-oriented activities is further used to identify tools and techniques that can assist software engineers create secure software as part of their day-to-day software development activities. In addition, the security threats and vulnerabilities are used to create catalogs that software engineers can use as quick references during the development of microservices. The proposed secure software development framework is validated by creating a microservices-based application.

The research conducted in this thesis is considered successful based on the research objectives and research questions formulated in Chapter 1. A software development framework of secure microservices has been created. The framework has been validated by developing an application using the artefacts of the framework. The contributions of this research were presented and opportunities for future research identified.

References

Aceto, G., Botta, A., De Donato, W. and Pescapè, A., 2013. Cloud monitoring: A survey. *Computer Networks*, *57*(9), pp.2093-2115.

Acetozi, J., 2017. Unit Tests. In *Pro Java Clustering and Scalability* (pp. 121-125). Apress, Berkeley, CA.

Ahmadvand, M. and Ibrahim, A., 2016, September. Requirements reconciliation for scalable and secure microservice (de) composition. In *Requirements Engineering Conference Workshops (REW), IEEE International* (pp. 68-73). IEEE.

Ahmadvand, M. and Ibrahim, A., 2016, September. Requirements reconciliation for scalable and secure microservice (de) composition. In *Requirements Engineering Conference Workshops (REW), IEEE International* (pp. 68-73). IEEE.

Albattah, W. and Melton, A., 2014, June. Package cohesion classification. *In Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on (pp. 1-8).* IEEE.

Alferez, G.H. and Pelechano, V., 2013, June. Facing uncertainty in web service compositions. In *Web Services (ICWS), 2013 IEEE 20th International Conference on* (pp. 219-226). IEEE.

AlHogail, A., 2015. Design and validation of information security culture framework. *Computers in Human Behavior*, *49*, pp.567-575.

Aljawarneh, S.A., Alawneh, A., and Jaradat, R., 2017. Cloud security engineering: Early stages of SDLC. *Future Generation Computer Systems*, *74*, pp.385-392.

Almorsy, M., Grundy, J., and Müller, I., 2016. An analysis of the cloud computing security problem. *arXiv preprint arXiv:1609.01107*.

Alpers, S., Becker, C., Oberweis, A. and Schuster, T., 2015, September. Microservice based tool support for business process modeling. *In Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International (pp. 71-78).* IEEE.

Anderson, C., 2015. Docker [software engineering]. *IEEE Software, 32(3), pp.102-c3.*

Andrews, K., Steinau, S. and Reichert, M., 2017, October. Enabling Fine-grained Access Control in Flexible Distributed Object-aware Process Management Systems. In *Enterprise Distributed Object Computing Conference (EDOC), 2017 IEEE 21st International* (pp. 143-152). IEEE.

AppDynamics, A.I.P., AppDynamics Pro Documentation.

Arce, I., Clark-Fisher, K., Daswani, N., DelGrosso, J., Dhillon, D., Kern, C., Kohno, T., Landwehr, C., McGraw, G., Schoenfield, B. and Seltzer, M., 2014. Avoiding the top 10 software security design flaws. *Technical report, IEEE Computer Society Center for Secure Design (CSD)*.

Arteau, P., Find security bugs, 2016. *URL http://find-sec-bugs. GitHub. io*.

Athanasopoulos, E., Boehner, M., Ioannidis, S., Giuffrida, C., Pidan, D., Prevelakis, V., Sourdis, I., Strydis, C. and Thomson, J., 2015, December. Secure hardware-software architectures for robust computing systems. In *International Conference on e-Democracy* (pp. 209-212). Springer, Cham.

Atkinson, B., Della-Libera, G., Hada, S., Hondo, M., Hallam-Baker, P., Klein, J., LaMacchia, B., Leach, P., Manferdelli, J., Maruyama, H. and Nadalin, A., 2002. Web services security (WS-Security). *Specification, Microsoft Corporation*.

Azarmi, M. and Bhargava, B., 2017, June. End-to-End Policy Monitoring and Enforcement for Service-Oriented Architecture. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on* (pp. 58-65). IEEE.

B. D. Win, R. Scandariato, K. Buyens, J. Grgoire, and W. Joosen. On the secure software development process: Clasp, {SDL} and touchpoints compared. Information and Software Technology, 51(7):1152 – 1171, 2009. Special Section: Software Engineering for Secure Systems Software Engineering for Secure Systems.

B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering," Tech. Rep. EBSE 2007- 001, Keele University and Durham University Joint Report, 2007.

B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering," Tech. Rep. EBSE 2007- 001, Keele University and Durham University Joint Report, 2007.

Baca, D. and Carlsson, B., 2011, May. Agile development with security engineering activities. In *Proceedings of the 2011 International Conference on Software and Systems Process*(pp. 149-158). ACM.

Bajaj, S., Box, D., Chappell, D., Curbera, F., Daniels, G., Hallam-Baker, P., Hondo, M., Kaler, C., Langworthy, D., Malhotra, A. and Nadalin, A., 2004. Web services policy framework (ws-policy). *Specification, IBM, BEA, Microsoft, SAP AG, Sonic Software, VeriSign*.

Balalaie, A., Heydarnoori, A. and Jamshidi, P., 2015. *Microservices migration patterns.* Tech. Rep. TR-SUTCE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, Tehran, Iran.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A. and Suter, P., 2017. Serverless Computing: Current Trends and Open Problems. *arXiv preprint arXiv:1706.03178.*

Baresi, L., 2017, December. Supporting the Decision of Migrating to Microservices Through Multi-layer Fuzzy Cognitive Maps. *In Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings (Vol. 10601, p. 471).* Springer.

Baresi, L., Mendonça, D.F. and Garriga, M., 2017, September. Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture. *In European Conference on Service-Oriented and Cloud Computing (pp. 196-210).* Springer, Cham.

Barnum, S. and Sethi, A., 2007. Attack patterns as a knowledge resource for building secure software. In *OMG Software Assurance Workshop: Cigital*.

Bartolini, C., Bertolino, A., Elbaum, S. and Marchetti, E., 2011. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software*, *84*(4), pp.655-668.

Bass, L., Holz, R., Rimba, P., Tran, A.B. and Zhu, L., 2015, May. Securing a deployment pipeline. In *Release Engineering (RELENG), 2015 IEEE/ACM 3rd International Workshop on* (pp. 4-7). IEEE.

Bass, L., Weber, I. and Zhu, L., 2015. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

Bean, James. *SOA and web services interface design: principles, techniques, and standards*. Morgan Kaufmann, 2009.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R. and Kern, J., 2001. The agile manifesto.

Behrens, S., Heffner J. (2017) The Avalanche Application DoS In Microservice Architectures. Available at: https://medium.com/signal-sciences-labs/starting-the-avalanche-application-dos-in-microservice-architectures-4f5eb4730a60 (Accessed 20 January 2019).

Behringer, M., Bjarnason, S., Jiang, S., Carpenter, B., Pritikin, M., Ciavaglia, L. and Clemm, A., 2015. Autonomic networking: Definitions and design goals.

Bell, M., 2008. *Service-oriented modeling (SOA): Service analysis, design, and architecture*. John Wiley & Sons.

ben Othmane, L., Angin, P. and Bhargava, B., 2014, September. Using assurance cases to develop iteratively security features using scrum. In Availability, Reliability and Security (ARES), 2014 Ninth International Conference on (pp. 490-497). IEEE.

ben Othmane, L., Angin, P., Weffers, H. and Bhargava, B., 2014. Extending the agile development process to develop acceptably secure software. *IEEE Transactions on dependable and secure computing*, *11*(6), pp.497-509.

ben Othmane, L., Angin, P., Weffers, H. and Bhargava, B., 2014. Extending the agile development process to develop acceptably secure software. *IEEE Transactions on dependable and secure computing*, *11*(6), pp.497-509.

Berg, A., 2012. *Jenkins Continuous Integration Cookbook*. Packt Publishing Ltd.
Bernstein, D., 2015. Is Amazon Becoming the New Cool Software Company for Developers? *IEEE Cloud Computing,* 2(1), pp.69-71.

Bertino, E., Martino, L.D., Paci, F. and Squicciarini, A.C., 2009. Web services threats, vulnerabilities, and countermeasures. In *Security for Web Services and Service-Oriented Architectures* (pp. 25-44). Springer, Berlin, Heidelberg.

Bertolino, A., Busch, M., Daoudagh, S., Lonetti, F. and Marchetti, E., 2014. A toolchain for designing and testing access control policies. In *Engineering Secure Future Internet Services and Systems* (pp. 266-286). Springer, Cham.

Bertolino, A., Busch, M., Daoudagh, S., Lonetti, F. and Marchetti, E., 2014. A toolchain for designing and testing access control policies. In *Engineering Secure Future Internet Services and Systems* (pp. 266-286). Springer, Cham.

Bhattacharya, K., Hull, R. and Su, J., 2009. A data-centric design methodology for business processes. *Handbook of Research on Business Process Modeling*, pp.503-531.

Bhuyan, P., Prakash, C. and Mohapatra, D., 2012. A survey of regression testing in SOA. *International Journal of Computer Applications*, *44*(19), pp.0975-8887.

Bigdoli, H., 2006. Handbook of Information Security, Key Concepts, Infrastructure, Standards, and Protocols.

Borazjani, P.N., 2017, May. Security Issues in Cloud Computing. In *International Conference on Green, Pervasive, and Cloud Computing* (pp. 800-811). Springer, Cham.

Bossert, O., 2016. A Two-Speed Architecture for the Digital Enterprise. In *Emerging Trends in the Evolution of Service-Oriented and Enterprise Architectures* (pp. 139-150). Springer International Publishing.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S. and Winer, D., 2004. Simple Object Access Protocol (SOAP) 1.1. W3C Note, May 2000. *W3C, http://www. w3. org/TR/2000/NOTE-SOAP-20000508*, pp.1-33.

Brewer, E.A., 2015, August. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (pp. 167-167). ACM.

Brodecki, B., Szychowiak, M. and Sasak, P., 2012. Security policy conflicts in service-oriented systems. *New Generation Computing*, *30*(2-3), pp.215-240.

Brook, J.M. and Brooks, R., 2015. A decade of lessons learned: Transforming the enterprise for today's cloud architecture. In *Proceedings of the ICCSM2015 3rd International Conference on Cloud Security and Management: ICCSM* (p. 16).

Brown, W.A., Holley, K.L., Moore, G.A. and Tegan, W.J., International Business Machines Corporation, 2014. *Defining service ownership for a service oriented architecture*. U.S. Patent 8,660,885.

Buecker, A., Ashley, P., Borrett, M., Lu, M., Muppidi, S. and Readshaw, N., 2008. *Understanding SOA security design and implementation*. IBM Redbooks.

Butzin, B., Golatowski, F. and Timmermann, D., 2016, September. Microservices approach for the internet of things. *In Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on (pp. 1-6).* IEEE.

Calabrese, J., Muñoz, R., Pasini, A., Esponda, S., Boracchia, M. and Pesado, P., 2017, October. Assistant for the Evaluation of Software Product Quality Characteristics Proposed by ISO/IEC 25010 Based on GQM-Defined Metrics. In *Argentine Congress of Computer Science* (pp. 164-175). Springer, Cham.

Campbell, G. and Papapetrou, P.P., 2013. *SonarQube in action*. Manning Publications Co.

Candido, G., Sousa, C., Di Orio, G., Barata, J. and Colombo, A.W., 2013, May. Enhancing device exchange agility in Service-oriented industrial automation. In *Industrial Electronics (ISIE), 2013 IEEE International Symposium on* (pp. 1-6). IEEE.

Casteele, S.V., 2005. Threat modeling for web application using the STRIDE model.

Chan, G.Y., Chua, F.F. and Lee, C.S., 2016. Intrusion detection and prevention of web service attacks for software as a service: Fuzzy association rules vs fuzzy associative patterns. *Journal of Intelligent & Fuzzy Systems*, *31*(2), pp.749-764.

Cherdantseva, Y. and Hilton, J., 2013, September. A reference model of information assurance & security. In *Availability, reliability, and security (ares), 2013 eighth international conference on* (pp. 546-555). IEEE.

Chinnici, R., Moreau, J.J., Ryman, A. and Weerawarana, S., 2007. Web services description language (wsdl) version 2.0 part 1: Core language. *W3C recommendation*, *26*, p.19.

Christensen, B., 2012. Introducing Hystrix for resilience engineering. *Netflix Tech Blog.*

Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S., 2001. Web Services Description Language (WSDL), W3C Note. *World Wide Web Consortium. URL: http://www. w3. org/TR/wsdl*.

Christensen, J.H., 2009, October. Using RESTful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications* (pp. 627-634). ACM.

Chrysikos, A. and McGuire, S., 2018. A Predictive Model for Risk and Trust Assessment in Cloud Computing: Taxonomy and Analysis for Attack Pattern Detection. In *Guide to Vulnerability Analysis for Computer Networks and Systems* (pp. 81-99). Springer, Cham.

Chung, S.K., Yee, O.C., Singh, M.M., and Hassan, R., 2014, September. SQL injections attack and session hijacking on e-learning systems. In *Computer, Communications, and Control Technology (I4CT), 2014 International Conference on* (pp. 338-342). IEEE.

Cloudmonix. 2018. CloudMonix. Retrieved 9-May-2018 from http://www. cloudmonix.com/

CloudWatch, A., 2014. Amazon cloudwatch.

Common Vulnerabilities Exposures, online: https://cve.mitre.org acessed on March 27, 2017.
Cramer, J. and Krueger, A.B., 2016. Disruptive change in the taxi business: The case of Uber. *American Economic Review*, *106*(5), pp.177-82.

Cramer, J. and Krueger, A.B., 2016. Disruptive change in the taxi business: The case of Uber. *American Economic Review*, *106*(5), pp.177-82.

Cruzes, D.S., Felderer, M., Oyetoyan, T.D., Gander, M. and Pekaric, I., 2017, May. How is Security Testing Done in Agile Teams? A Cross-Case Analysis of Four Software Teams. In *International Conference on Agile Software Development* (pp. 201-216). Springer, Cham.

Da Xu, L., 2014. *Enterprise integration and information architecture*. CRC Press.

Daya, S., Van Duy, N., Eati, K., Ferreira, C.M., Glozic, D., Gucer, V., Gupta, M., Joshi, S., Lampkin, V., Martins, M. and Narain, S., 2016. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks.

de Andrade Gomes, P.H., Garcia, R.E., Spadon, G., Eler, D.M., Olivete, C., and Correia, R.C.M., 2017, October. Teaching software quality via source code inspection tool. In *2017 IEEE Frontiers in Education Conference (FIE)* (pp. 1-8). IEEE.

de Andrade Gomes, P.H., Garcia, R.E., Spadon, G., Eler, D.M., Olivete, C., and Correia, R.C.M., 2017, October. Teaching software quality via source code inspection tool. In *2017 IEEE Frontiers in Education Conference (FIE)* (pp. 1-8). IEEE.

De Giorgio, T., Ripa, G. and Zuccalà, M., 2010. An approach to enable replacement of SOAP services and REST services in lightweight processes. *Current Trends in Web Engineering*, pp.338-346

Della-Libera, G., Dixon, B., Farrell, J., Garg, P., Hondo, M., Kaler, C., Lampson, B., Lawrence, K., Layman, A., Leach, P. and Manferdelli, J., 2002. Security in a web services world: A proposed architecture and roadmap. *White paper, IBM Corporation, and Microsoft Corporation*.

Dell'Amico, M., Serme, G., Idrees, M.S., De Oliveira, A.S. and Roudier, Y., 2013. Hipolds: a hierarchical security policy language for distributed systems. *Information Security Technical Report*, *17*(3), pp.81-92.

Dhara, K.M., Dharmala, M. and Sharma, C.K., 2015. A Survey Paper on Service Oriented Architecture Approach and Modern Web Services.

Di Francesco, P., Malavolta, I. and Lago, P., 2017, April. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *Software Architecture (ICSA), 2017 IEEE International Conference on* (pp. 21-30). IEEE.

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L., 2016. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*.

Ebert, C., Gallardo, G., Hernantes, J. and Serrano, N., 2016. DevOps. *IEEE Software*, *33*(3), pp.94-100.

El Hassani, A.A., El Kalam, A.A., Bouhoula, A., Abassi, R. and Ouahman, A.A., 2015. Integrity-OrBAC: a new model to preserve Critical Infrastructures integrity. *International journal of information security*, *14*(4), pp.367-385.

Elsen, R., Liem, I. and Akbar, S., 2016, October. Software versioning quality parameters: Automated assessment tools based on the parameters. In *Data and Software Engineering (ICoDSE), 2016 International Conference on* (pp. 1-6). IEEE.

Erl Thomas, Service-Oriented Architecture: Concepts, Technology, and Design. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

Erl, T., 2008. *Soa: principles of service design* (Vol. 1). Upper Saddle River: Prentice Hall.

Erl, T., Gee, C., Kress, J., Maier, B., Normann, H., Raj, P., Shuster, L., Trops, B., Utschig-Utschig, C., Wik, P. and Winterberg, T., 2014. Next Generation SOA.

Erl, T., Merson, P. and Stoffers, R., 2017. *Service-oriented Architecture: Analysis and Design for Services and Microservices*. Prentice Hall PTR.

Estdale, J. and Georgiadou, E., 2018, September. Applying the ISO/IEC 25010 Quality Models to Software Product. In *European Conference on Software Process Improvement* (pp. 492-503). Springer, Cham.

Farrell, S., 2009. API Keys to the Kingdom. *IEEE Internet Computing*, *13*(5).

Fatema, K., Emeakaroha, V.C., Healy, P.D., Morrison, J.P. and Lynn, T., 2014. A survey of Cloud monitoring tools: Taxonomy, capabilities, and objectives. *Journal of Parallel and Distributed Computing*, *74*(10), pp.2918-2933.

Feitelson, D.G., Frachtenberg, E. and Beck, K.L., 2013. Development and deployment at facebook. *IEEE Internet Computing, 17(4)*, pp.8-17.

Feng, Q., Kazman, R., Cai, Y., Mo, R. and Xiao, L., 2016, April. Towards an architecture-centric approach to security analysis. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)* (pp. 221-230). IEEE.

Fernandez, E.B., Astudillo, H. and Pedraza-García, G., 2015, September. Revisiting architectural tactics for security. In *European Conference on Software Architecture* (pp. 55-69). Springer, Cham.

Fernández, L., Hagenrud, H., Mudingay, R., Korhonen, T., Zupanc, B. and Andersson, R., 2016. How to Build and Maintain a Development Environment for the Development of Controls Software Applications: An Example of" Infrastructure as Code" within the Physics Accelerator Community.

Fernandez-Buglioni, E., 2013. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons.

Fetzer, C., 2016. Building critical applications using microservices. *IEEE Security & Privacy*, (6), pp.86-89.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T., 1999. *Hypertext transfer protocol--HTTP/1.1* (No. RFC 2616).

Firesmith, D., 2004. Specifying reusable security requirements. *Journal of Object Technology*, *3*(1), pp.61-75.

Fisher, J., Koning, D. and Ludwigsen, A.P., 2013, October. Utilizing atlassian jira for large-scale software development management. In *14th International Conference on Accelerator & Large Experimental Physics Control Systems (ICALEPCS)*.

Fontdevila, D., Genero, M. and Oliveros, A., 2017, November. Towards a usability model for software development process and practice. In *International Conference on Product-Focused Software Process Improvement* (pp. 137-145). Springer, Cham.

Forsgren Velasquez, N., Kim, G., Kersten, N. and Humble, J., 2017. State of DevOps report. *Puppet Labs and IT Revolution*.

Fulton III, S.M., 2015. "What led amazon to its own microservices architecture.
G. McGraw, Software Security: Building Security, Addison Wesley, 2006

Geer, D., 2010. Are companies actually using secure development life cycles?. *Computer*, *43*(6), pp.12-16.

Giarratano, D., Guise, L., and Bodin, J.Y., 2017. Does cyber security moving towards risk management leads to new grid organisation?. *CIRED-Open Access Proceedings Journal*, *2017*(1), pp.2700-2702.

Giles, M. (2017) 'Uber Paid Off Hackers to Hide Massive Data Breach,' MIT Technology Review, 22 November [Online]. Available at: https://www.technologyreview.com/s/609539/uber-paid-off-hackers-to-hide-massive-data-breach/(Accessed: 8 January 2018).

Gkioulos, V. and Wolthusen, S.D., 2017, August. Security Requirements for the Deployment of Services Across Tactical SOA. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security* (pp. 115-127). Springer, Cham.

Gogouvitis, S.V., Alexandrou, V., Mavrogeorgi, N., Koutsoutos, S., Kyriazis, D. and Varvarigou, T., 2012, November. A monitoring mechanism for storage clouds. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*(pp. 153-159). IEEE.

Gressin, S., 2017. The Equifax Data Breach: What to Do.

Guide, D., 2010. Amazon Elastic Load Balancing.

Gummaraju, J., Desikan, T. and Turner, Y., 2015. Over 30% of official images in docker hub contain high priority security vulnerabilities. *Https://Banyanops. Com*, pp.1-6.

Gutierrez, F., 2016. Pro Spring Boot. Apress.

Gutierrez, F., 2017. AMQP with Spring Boot. In *Spring Boot Messaging* (pp. 59-80). Apress.

Haley, C., Laney, R., Moffett, J. and Nuseibeh, B., 2008. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering*, *34*(1), pp.133-153.

Hall, D., 2013. *Ansible configuration management*. Packt Publishing Ltd.

Hall, J., 2016. *Mastering SaltStack*. Packt Publishing Ltd.

Härlin, M., 2016. Testing and Gherkin in agile projects.

Hawanna, V., Kulkarni, V., Rane, R. and Joshi, P., 2016, December. Risk Evaluation of X. 509 Certificates–A Machine Learning Application. In *International Conference on Information Systems Security* (pp. 372-389). Springer, Cham.

He, X. and Yang, X., 2017, October. Authentication and Authorization of End User in Microservice Architecture. *In Journal of Physics: Conference Series (Vol. 910, No. 1, p. 012060).* IOP Publishing.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L.E., Pahl, C., Schulte, S. and Wettinger, J., 2017, April. Performance engineering for microservices: research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (pp. 223-226). ACM.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L.E., Pahl, C., Schulte, S. and Wettinger, J., 2017, April. Performance engineering for microservices: research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (pp. 223-226). ACM.

Highsmith, J., 2010. Beyond scope, schedule, and cost: The agile triangle. *Dostupno na: http://jimhighsmith. com/2010/11/14/beyond-scope-schedule-andcost-the-agile-triangle*.

Hilton, M., Tunnell, T., Huang, K., Marinov, D. and Dig, D., 2016, August. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (pp. 426-437). ACM.

Hochstein, L. and Moser, R., 2017.*Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. " O'Reilly Media, Inc.."

Hoffmann, J. and Weber, I., 2014. Web Service Composition. In *Encyclopedia of Social Network Analysis and Mining* (pp. 2389-2399). Springer New York.

Hohpe, G. and Woolf, B., 2004. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

Howard, M. and LeBlanc, D., Writing Secure Code. 2003.

Howard, M. and Lipner, S., 2006. The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software (Developer Best Practices).

Iacono, L.L. and Nguyen, H.V., 2015, June. Authentication scheme for REST. In *International Conference on Future Network Systems and Security* (pp. 113-128). Springer, Cham.
IEC, I., 2013. 27002: 2013. *Information technology Security techniques-Code of pract ice for information security controls. Retrieved from http://www. iso. org/iso/catalogue_detail*.
Inci, M.S., Gülmezoglu, B., Apecechea, G.I., Eisenbarth, T. and Sunar, B., 2015. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. IACR Cryptology ePrint Archive, 2*015*, p.898.

Infoholic Research LLP. Microservice Architecture Market: Global Drivers, Restraints, Opportunities, Trends, and Forecasts up to 2023. Sept. 2017. url: https://www.researchandmarkets.com/research/szk939/microservice (visited on May 21, 2018)

IntelliJ, I.D.E.A., 2011. the most intelligent Java IDE. *JetBrains[online].[cit. 2016-02-23]*.

ISO, I., 1989. 7498-2. information processing systems open systems interconnection basic reference model-part 2: Security architecture. *ISO Geneva, Switzerland*.

ISO, I., 7498-2: 1989. *Information processing systems-Open Systems Interconnection*, pp.7498-2.

Jain, S. and Ingle, M., 2011. Software security requirements gathering instrument. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *2*(7), pp.116-129.

Jaramillo, D., Nguyen, D.V. and Smart, R., 2016, March. Leveraging microservices architecture by using Docker technology. *In SoutheastCon, 2016 (pp. 1-5).* IEEE.

Jessani, V., Iyengar, A. and Chilanti, M., 2007. WebSphere business integration primer: Process server, BPEL, SCA, and SOA. Pearson Education.

Johansson, P., 2017. Efficient Communication With Microservices.

Jones, M. and Hildebrand, J., 2015. *Json web encryption (jwe)*(No. RFC 7516).

Jones, M., Bradley, J., and Sakimura, N., 2015. *JSON web token (jwt)* (No. RFC 7519).

Jøsang, A., Ødegaard, M. and Oftedal, E., 2015, May. Cybersecurity Through Secure Software Development. In *IFIP World Conference on Information Security Education* (pp. 53-63). Springer, Cham.

Jose, A., and Shettar, R., 2017. Cloud deployments using micro-services for digital monetization application for customers. *IJITR, 5(3),* pp.6492-6495.

Jurimae, S., 2015. *A Literature Survey of the Development Processes for Secure Software* (Doctoral dissertation, Bachelor's Thesis, Faculty of Mathematics and Computer Science, University of Tartu).

Kadam, S.P. and Joshi, S., 2015, March. Secure by design approach to improve the security of object-oriented software. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on* (pp. 24-30). IEEE.

Kakavand, M., Mustapha, N., Mustapha, A., Abdullah, M.T. and Ahmadi, B., 2016. Towards a Defense Mechanism Against REST-Based Web Service Attacks. *Advanced Science Letters*, *22*(10), pp.2827-2831.

Kanneganti, R. and Chodavarapu, P., 2008. *SOA security*. Dreamtech Press.
Kaur, H., 2017. Automating Static Code Analysis for Risk Assessment and Quality Assurance of Medical Record Software

Kaur, H., 2017. Automating Static Code Analysis for Risk Assessment and Quality Assurance of Medical Record Software.

Khaim, R., Naz, S., Abbas, F., Iqbal, N., Hamayun, M. and Pakistan, R., 2016. A review of security integration technique in agile software development. *International Journal of Software Engineering & Applications*, *7*(3).

Killalea, T., 2016. The hidden dividends of microservices. *Communications of the ACM, 59(8),* pp.42-45.

Kim, G., 2014. DevOps Patterns Distilled: A Fifteen Year Study of High Performing {IT} Organizations.

Kissel, R., 2013. Glossary of key information security terms. *NIST Interagency Reports NIST IR*, *7298*(3).

König, L. and Steffens, A., 2018. Towards a Quality Model for DevOps. *Continuous Software Engineering & Full-scale Software Engineering*, p.37.

Krafzig D, Banke K, Slama D. Enterprise SOA: service-oriented architecture best practices. Prentice Hall Professional; 2005.

Kravchuk, S., Minochkin, D., Omiotek, Z., Bainazarov, U., Weryńska-Bieniasz, R. and Iskakova, A., 2017, August. Cloud-based mobility management in heterogeneous wireless networks. *In Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2017 (Vol. 10445, p. 104451W).* International Society for Optics and Photonics.

Krivic, P., Skocir, P., Kusek, M. and Jezic, G., 2017, June. Microservices as Agents in IoT Systems. *In KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications (pp. 22-31).* Springer, Cham.

Kuusela, J., 2017. Security testing in continuous integration processes.

Lalsing, V., Kishnah, S. and Pudaruth, S., 2012. People factors in agile software development and project management. *International Journal of Software Engineering & Applications, 3(1),* p.117.

Lawton, G., 2015. TechTarget How microservices bring agility to SOA.

Le Ru, Y., Aron, M., Gerval, J.P. and Napoleon, T., 2015. Tests generation oriented web-based automatic assessment of programming assignments. *Smart education and smart e-learning* (pp. 117-127). Springer, Cham.

.

Le, V.D., Neff, M.M., Stewart, R.V., Kelley, R., Fritzinger, E., Dascalu, S.M. and Harris, F.C., 2015, July. Microservice-based architecture for the NRDC. *In Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on (pp. 1659-1664).* IEEE.

Lee, J.H., Shim, H.J. and Kim, K.K., 2010. Critical success factors in SOA implementation: an exploratory study. *Information systems management*, *27*(2), pp.123-145.

Lesser, E. and Ban, L., 2016. How leading companies practice software development and delivery to achieve a competitive edge. *Strategy & Leadership*, *44*(1), pp.41-47.

Lewis, G.A., Morris, E., Simanta, S. and Wrage, L., 2007, February. Common misconceptions about service-oriented architecture. In *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2007. ICCBSS'07. Sixth International IEEE Conference on* (pp. 123-130). IEEE.

LightStep, Inc. The 2018 Global Microservice Trends report. Apr. 2018. url: https://go.lightstep.com/global-microservices-trends-report-2018 (visited on May 21, 2018) (cit. on p. 16).

Linthicum, D.S., 2003. Next generation application integration: from simple information to web services. Addison-Wesley Longman Publishing Co., Inc.

Litwin, W., Mark, L. and Roussopoulos, N., 1990. Interoperability of multiple autonomous databases. *ACM Computing Surveys (CSUR)*, *22*(3), pp.267-293.

Liu, J., Gu, N., Zong, Y., Ding, Z. and Zhang, Q., 2005, September. Service registration and discovery in a domain-oriented UDDI registry. In *Computer and Information Technology, 2005. CIT 2005. The Fifth International Conference on* (pp. 276-283). IEEE.

Long, F., Mohindra, D., Seacord, R.C., Sutherland, D.F. and Svoboda, D., 2011. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional.

Long, S.J., 2015. Owasp dependency check.

Loope, J., 2011. *Managing Infrastructure with Puppet: Configuration Management at Scale*. " O'Reilly Media, Inc.."

Loukides, M., 2012. *What is DevOps?*. " O'Reilly Media, Inc.".

M. Howard, S. Lipner, The Security Development Lifecycle (SDL): A Process for Developing Demonstrably More Secure Software, Microsoft Press, 2006.

M. Richards, "Microservices vs. service-oriented architecture," 2015. O'Reilly Media

Macero, M., 2017. The Microservices Journey Through Tools. *In Learn Microservices with Spring Boot (pp. 179-265)*. Apress, Berkeley, CA.

MacLennan, E. and Van Belle, J.P., 2014. Factors affecting the organizational adoption of service-oriented architecture (SOA). *Information Systems and e-Business Management*, *12*(1), pp.71-100.

Mardjan, M.J. and Jahan, A., 2016. Open Reference Architecture for Security and Privacy.
Marks, Eric A., and Michael Bell. *Service Oriented Architecture (SOA): a planning and implementation guide for business and technology*. John Wiley & Sons, 2008.

Markus, M.L. and Tanis, C., 2000. The enterprise systems experience-from adoption to success. *Framing the domains of IT research: Glimpsing the future through the past*, *173*, pp.207-173
Marschall, M., 2015. *Chef infrastructure automation cookbook*. Packt Publishing Ltd.

Martins, G., Bhatia, S., Koutsoukos, X., Stouffer, K., Tang, C. and Candell, R., 2015, August. Towards a systematic threat modeling approach for cyberphysical systems. In *Resilience Week (RWS), 2015* (pp. 1-6). IEEE.

McGrath, G. and Brenner, P.R., 2017, June. Serverless computing: Design, implementation, and performance. *In Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on (pp. 405-410)*. IEEE.

Menasce, D.A., 2005. Mom vs. rpc: Communication models for distributed applications. *IEEE Internet Computing*, *9*(2), pp.90-93.

Merkel, D., 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal, 2014(239)*, p.2.

Moller and Schwartzbach 2006. *An introduction to XML and Web technologies*. Pearson Education.

Montesi, F. and Weber, J., 2016. Circuit breakers, discovery, and API gateways in microservices. *arXiv preprint arXiv:1609.05830*.

Morgan, S., 2018. Top 5 cybersecurity facts, figures and statistics for 2018 (2018).

Myagmar, S., Lee, A. J., and Yurcik, W., 2005. Threat modeling as a basis for security requirements, in Symposium on requirements engineering for information security (SREIS), vol. 2005, 2005, pp. 1–8.

Myers, C., 2016. *Learning saltstack*. Packt Publishing Ltd.

Nacer, H., Djebari, N., Slimani, H., and Aissani, D., 2017. A distributed authentication model for composite Web services. *Computers & Security*, *70*, pp.144-178.

Nadalin, A., Kaler, C., Monzillo, R. and Hallam-Baker, P., 2006. Web services security: SOAP message security 1.1 (WS-Security 2004). *Oasis Standard*, *200401*.

Nadareishvili, I., Mitra, R., McLarty, M. and Amundsen, M., 2016. *Microservice Architecture: Aligning Principles, Practices, and Culture.* " O'Reilly Media, Inc.".

Natis, Y. and Schulte, R., 2003. Introduction to service-oriented architecture. *Gartner Group*, *14*.

Netflix (2012). The Netflix Tech Blog: Netflix Shares Cloud Load Balancing And Failover Tool: Eureka!

Neumann, P.G., 2018. Fundamental trustworthiness principles. *New Solutions for Cybersecurity*.
.

Newcomer, E. (2017) 'Uber Paid Hackers to Delete Stolen Data on 57 Million People',

Newcomer, E. and Lomow, G., 2005. *Understanding SOA with Web services*. Addison-Wesley.

Newman, S., 2015. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

Ni, C.Y. and Yuan, S.M., 1996. DITSE: an experimental distributed database system. *Information and Software Technology*, *38*(2), pp.103-110.

Obe, R.O. and Hsu, L.S., 2017. *PostgreSQL: Up and Running: a Practical Guide to the Advanced Open Source Database*. " O'Reilly Media, Inc.".

Oberscheven, F.M., 2013. Software Quality Assessment in an Agile Environment. *Faculty of Science of Radboud University in Nijmegen.*

O'Brien, S.A., 2017. Giant Equifax data breach: 143 million people could be affected. *CNN Tech*.

Ochieng, D., Gichoya, D. and Odini, C., 2011, May. Proposed ICT-enabled services model for local authorities in Kenya. In *IST-Africa Conference Proceedings, 2011* (pp. 1-9). IEEE.

Oftedal, A.V.D.S.E. and Stock, A., 2014. REST Security Cheat Sheet.
Olivier, M.S., 2009. *Information technology research: A practical guide for computer science and informatics*. Pretoria: Van Schaik.

Olsson, R.A. and Keen, A.W., 2004. Remote procedure call. The JR Programming Language: Concurrent Programming in an Extended Java, pp.91-105.

Open Security Alliance (2017), "IT Security Requirements" [Online]. [Accessed 15 January 2018], available:
http://www.opensecurityarchitecture.org/cms/definitions/it_security_requirements

Open Web Application Security Project, "OWASP Threat Dragon,"
https://www.owasp.org/index.php/OWASP Threat Dragon, 2018.

Otterstad, C. and Yarygina, T., 2017, September. Low-level exploitation mitigation by diverse microservices. In *European Conference on Service-Oriented and Cloud Computing* (pp. 49-56). Springer, Cham.

OWASP Foundation: OWASP Secure Coding Practices Quick Reference Guide V2. [Online].
https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf (2010)

OWASP, Comprehensive, lightweight application security process, http://www.owasp.org, 2006.

Oyetoyan, T.D., Cruzes, D.S. and Jaatun, M.G., 2016, August. An empirical study on the relationship between software security skills, usage and training needs in agile settings. In *2016 11th International Conference on Availability, Reliability and Security (ARES)* (pp. 548-555). IEEE.

Pahl, C. and Jamshidi, P., 2016, April. Microservices: A Systematic Mapping Study. In *CLOSER (1)* (pp. 137-146).

Pant, K. and Juric, M.B., 2008. Business process driven SOA using BPMN and BPEL: From business process modeling to orchestration and service oriented architecture. Packt Publishing Ltd.

Paul, M., 2016. *Official (ISC) 2 Guide to the CSSLP*. CRC Press.

Pautasso, C., 2014. RESTful web services: principles, patterns, emerging technologies. In *Web Services Foundations* (pp. 31-51). Springer New York.

Peischl, B., Felderer, M. and Beer, A., 2016, August. Testing security requirements with non-experts: approaches and empirical investigations. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on* (pp. 254-261). IEEE.

Penzenstadler, B., Raturi, A., Richardson, D. and Tomlinson, B., 2014. Safety, security, now sustainability: The nonfunctional requirement for the 21st century. *IEEE Software*, *31*(3), pp.40-47.

Petersen, K., Feldt, R., Mujtaba, S. and Mattsson, M., 2008, June. Systematic Mapping Studies in Software Engineering. In *EASE* (Vol. 8, pp. 68-77).

Popa, C.E., 2015. Securing a REST Web Service. *Journal of Mobile, Embedded and Distributed Systems*, *7*(2), pp.95-99.

Porter, M.E. and Heppelmann, J.E., 2015. How smart, connected products are transforming companies. *Harvard Business Review*, *93*(10), pp.96-114.

Priya, S.S., and Arya, S.S., 2016. Threat Modeling for a Secured Software Development. *International Journal of Advanced Research in Computer Science*, *7*(1).

Rahman, M. and Gao, J., 2015, March. A reusable automated acceptance testing architecture for microservices in behavior-driven development. I*n Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on (pp. 321-325).* IEEE.

Raman, R.C. and Dewailly, L., 2018. *Building RESTful Web Services with Spring 5: Leverage the power of Spring 5.0, Java SE 9, and Spring Boot 2.0*. Packt Publishing Ltd.

Rauter, T., Kajtazovic, N. and Kreiner, C., 2016. Asset-centric security risk assessment of software components. In *2nd International Workshop on MILS: Architecture and Assurance for Secure Systems*.

Ravichandran, A., Taylor, K. and Waterhouse, P., 2016. DevOps foundations. *In DevOps for Digital Leaders (pp. 27-47).* Apress.

Ray, K., 2010. Introduction to Service-Oriented Architectures. URL: http://anengineersperspective. com/wp-content/uploads/2010/03/Introduction-to-SOA. Pdf.

Reddy, K.S.P., 2017. Testing Spring Boot Applications. In *Beginning Spring Boot 2* (pp. 221-246). Apress, Berkeley, CA.

Richardson, C., 2016. Building Microservices: Using an API Gateway.

Richter, D., Konrad, M., Utecht, K. and Polze, A., 2017, July. Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice. *In Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on (pp. 130-137).* IEEE.
Rogers, B., 2015. The social costs of Uber. *U. Chi. L. Rev. Dialogue*, *82*, p.85.

Ross, R., McEvelley, M., and Oren, J.C., 2016. NIST special Publication 800-160 Systems Security Engineering-Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems. *Gaithersburg: National Institute of Standards and Technology*.

Rotter, C., Illés, J., Nyíri, G., Farkas, L., Csatári, G. and Huszty, G., 2017, March. Telecom strategies for service discovery in microservice environments. *In Innovations in Clouds, Internet and Networks (ICIN), 2017 20th Conference on (pp. 214-218).* IEEE.

Sahu, D.R., and Tomar, D.S., 2017. Analysis of Web Application Code Vulnerabilities using Secure Coding Standards. *Arabian Journal for Science and Engineering*, *42*(2), pp.885-895.

Sandkuhl, K. and Söderström, E., 2016. PoEM Doctoral Consortium: Proceedings of the Doctoral Consortium at the 9th IFIP WG 8.1 Working Conference on The Practice of Enterprise Modeling (PoEM-DC 2016), Skövde, Sweden, November 8th, 2016. *In The Practice of Enterprise Modeling (PoEM).* CEUR-WS. Org.

Santos, J.C., Tarrit, K. and Mirakhorli, M., 2017, April. A Catalog of Security Architecture Weaknesses. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on* (pp. 220-223). IEEE.

Santos, J.C., Tarrit, K., Sejfia, A., Mirakhorli, M. and Galster, M., 2019. An empirical study of tactical vulnerabilities. *Journal of Systems and Software*, *149*, pp.263-284.

Satoh, F. and Tokuda, T., 2011. Security policy composition for composite web services. *IEEE Transactions on Services Computing*, *4*(4), pp.314-327.

Satoh, F., Nakamura, Y., Mukhi, N.K., Tatsubori, M. and Ono, K., 2008, July. Methodology and tools for end-to-end soa security configurations. In *Services-Part I, 2008. IEEE Congress on* (pp. 307-314). IEEE.

Sbarski, P. and Kroonenburg, S., 2017. Serverless Architectures on AWS: With examples using AWS Lambda.

Scandariato, R., Wuyts, K., and Joosen, W., 2015. A descriptive study of Microsoft's threat modeling technique. *Requirements Engineering*, *20*(2), pp.163-180.

Schmidt, C., 2016. Agile Software Development. In *Agile Software Development Teams* (pp. 7-35). Springer, Cham.

Scott, B., Xu, J., Zhang, J., Brown, A., Clark, E., Yuan, X., Yu, A. and Williams, K., 2017, October. An interactive visualization tool for teaching ARP spoofing attack. In *2017 IEEE Frontiers in Education Conference (FIE)* (pp. 1-5). IEEE.

Seacord, R. and Sebor, M., 2010. Top 10 secure coding practices. *CERT, February*, *13*.
Shah, D. and Patel, D., 2008, September. Dynamic and ubiquitous security architecture for global SOA. In *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM'08. The Second International Conference on*(pp. 482-487). IEEE.

Shashwat, A., Kumar, D. and Chanana, L., 2017, December. An end to end security framework for service oriented architecture. In *Infocom Technologies and Unmanned Systems (Trends and Future Directions)(ICTUS), 2017 International Conference on* (pp. 475-480). IEEE.

Sheffer, Y., Holz, R. and Saint-Andre, P., 2015. *Summarizing known attacks on transport layer security (tls) and datagram tls (dtls)* (No. RFC 7457).

Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S. and Xu, X., 2014. Web services composition: A decade's overview. *Information Sciences*, *280*, pp.218-238.

Shepperd, M. and Ince, D., 1993. *Derivation and Validation of Software Metrics. International Series of Monographs on Computer Science*. Clarendon Press.

Shetty, R., Choo, K.K.R. and Kaufman, R., 2017, June. Shellshock Vulnerability Exploitation and Mitigation: A Demonstration. In *International Conference on Applications and Techniques in Cyber Security and Intelligence* (pp. 338-350). Edizioni della Normale, Cham.

Shostack, A., 2008, September. Experiences threat modeling at Microsoft. In *Modeling Security Workshop. Dept. of Computing, Lancaster University, UK*.

Shostack, A., 2014. *Threat modeling: Designing for security*. John Wiley & Sons.

Shuaibu, B.M., Norwawi, N.M., Selamat, M.H. and Al-Alwani, A., 2015. Systematic review of web application security development model. *Artificial Intelligence Review*, *43*(2), pp.259-276.

Sittig, D.F. and Singh, H., 2016. A socio-technical approach to preventing, mitigating, and recovering from ransomware attacks. *Applied clinical informatics*, *7*(2), p.624.

Souag, A., Mazo, R., Salinesi, C. and Comyn-Wattiau, I., 2016. Reusable knowledge in security requirements engineering: a systematic mapping study. *Requirements Engineering*, *21*(2), pp.251-283.

Stanek, M., 2017. Secure by default-the case of TLS. *arXiv preprint arXiv:1708.07569*.

Storms, A., 2015. How security can be the next force multiplier in devops. *RSAConference,(San Francisco, USA)*, pp.19-20.

Sun, Y., Nanda, S. and Jaeger, T., 2015, November. Security-as-a-service for microservices-based cloud applications. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on* (pp. 50-57). IEEE.

Sun, Y., Nanda, S. and Jaeger, T., 2015, November. Security-as-a-service for microservices-based cloud applications. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on* (pp. 50-57). IEEE.

Suzic, B., Prünster, B., Ziegler, D., Marsalek, A. and Reiter, A., 2016, October. Balancing utility and security: Securing cloud federations of public entities. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"* (pp. 943-961). Springer, Cham.

Tang, B., Sandhu, R., and Li, Q., 2015. Multi-tenancy authorization models for collaborative cloud services. *Concurrency and Computation: Practice and Experience*, *27*(11), pp.2851-2868.

Taspolatoglu, E. and Heinrich, R., 2016, April. Context-based architectural security analysis. In *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on* (pp. 281-282). IEEE.

Taylor, M. and Vargo, S., 2014. *Learning Chef: A Guide to Configuration Management and Automation*. " O'Reilly Media, Inc.."

Thönes, J., 2015. Microservices. *IEEE Software, 32(1),* pp.116-116.

Tian-yang G, Yin-Sheng S, You-yuan F. Research on software security testing. World Academy of science, engineering and Technology. 2010 Sep 21;70:647-51.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. and Edmonds, A., 2015, April. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud* (pp. 19-24). ACM.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. and Edmonds, A., 2015, April. An Top Threats Working Group, 2017. The Treacherous 12: Cloud Computing Top Threats in 2016. *Cloud Security Alliance. online at https://downloads. cloudsecurityalliance. org/assets/research/topthreats/Treacherous12_CloudComputing_TopThreats. pdf. Accessed*, *1*.

Tuma, K., Scandariato, R., Widman, M. and Sandberg, C., 2017. Towards Security Threats that Matter. In *Computer Security* (pp. 47-62). Springer, Cham.

Ulltveit-Moe, N. and Oleshchuk, V., 2015. A novel policy-driven reversible anonymization scheme for XML-based services. *Information Systems*, *48*, pp.164-178.

Uzunov, A.V., Fernandez, E.B. and Falkner, K., 2012. Engineering Security into Distributed Systems: A Survey of Methodologies. *J. UCS*, *18*(20), pp.2920-3006.

Varanasi, B. and Belida, S., 2015. Restful spring. In *Spring REST* (pp. 31-46). Apress, Berkeley, CA.

Vernadat, F.B., 2007. Interoperable enterprise systems: Principles, concepts, and methods. *Annual reviews in Control*, *31*(1), pp.137-145.

Videla, A. and Williams, J.J., 2012. RabbitMQ in action. *Manning*.

Viega, J., 2011. Ten Years of Trustworthy Computing: Lessons Learned. *IEEE Security & Privacy*, *9*(5), pp.3-4.

Wagner, B. and Sood, A., 2016, August. Economics of Resilient Cloud Services. *In Software Quality, Reliability and Security Companion (QRS-C), 2016 IEEE International Conference on (pp. 368-374)*. IEEE.

White, G.K., 2015. *Secure Coding Practices, Tools, and Processes* (No. LLNL-CONF-671591). Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

White, S.A. and Bock, C., 2011. BPMN 2.0 Handbook Second Edition: Methods, Concepts, Case Studies and Standards in Business Process Management Notation. Future Strategies Inc.

Williams, P.A., and McCauley, V., 2016, December. Always connected: The security challenges of the healthcare Internet of Things. In *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on* (pp. 30-35). IEEE.

Willnecker, F., Brunnert, A., Gottesheim, W. and Krcmar, H., 2015, January. Using dynatrace monitoring data for generating performance models of java ee applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (pp. 103-104). ACM.

Wolff, E., 2016. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional.
World Wide Web Consortium, W3C: Universal Description, Discovery and Integration: UDDI Spec Technical Committee Specification v. 3.0, (2003).

Woschek, M., 2015. OWASP Cheat Sheets.

Yamany, H.F.E., Capretz, M.A. and Allison, D.S., 2010. Intelligent security and access control framework for service-oriented architecture. *Information and Software Technology*, *52*(2), pp.220-236.

Yarygina T, Bagge AH. Overcoming security challenges in microservice architectures. InService-Oriented System Engineering (SOSE), 2018 IEEE Symposium on 2018 Mar 26 (pp. 11-20). IEEE.

Yarygina, T., 2018. Exploring Microservice Security.

Ye, W., 2013. *Instant Cucumber BDD How-to*. Packt Publishing Ltd.

Zabbix, S.I.A., 2014. Zabbix. The Enterprise-class Monitoring Solution for Everyone.

Zimmermann, O., 2015. Do microservices pass the same old architecture test? Or: Soa is not dead-long live (micro-) services. In *Microservices Workshop at SATURN Conference, SEI*.
Zimmermann, O., 2017. Microservices tenets. *Computer Science-Research and Development*, *32*(3-4), pp.301-310.

Zimmermann, O., 2017. Microservices tenets: agile approach to service development and deployment. *Computer Science-Research and Development, 32(3),* pp.301-310.

Zúñiga-Prieto, M., Insfran, E., Abrahao, S. and Cano-Genoves, C., 2016. Incremental Integration of Microservices in Cloud Applications.

Appendix A

Appendix A provides the security artefacts produced in the SAFEMicroservices preliminary phases.

(a) A.2.2 Microservice architecture common weakness enumeration

Table A.2.2 below is an artefact produced in the preliminary phase of SAFEMicroservices. The table contains a list of common vulnerabilities associated with each microservices threat. Software engineers can use this artefact to quickly identify vulnerabilities associated with each threat and the architectural concept that is the root cause of the vulnerabilities in microservices.

Table A.2.2 Microservices architecture common weakness enumeration

		Security threats				
Architecture category	**Common vulnerabilities**	**Insecure API**	**Unauthorized access**	**Insecure microservices discovery**	**Insecure runtime infrastructure**	**Insecure message broker**
Validate Input	• Improper input validation (CWE-20)	√	x	√	x	x
	• Improper neutralization of request data (CWE-138,150, 643,74,76,77,78,943,95,96, 93)	√	x	√	x	x
	• Acceptance of extraneous untrusted data with trusted data	√	x	√	x	√
	• Cross-site request forgery (CSRF) (CWE-352)	√	x	x	x	√
	• Deserialization of untrusted data (CWE-502)	√	x	√	x	√
	• Failure to sanitize special elements in request data (CWE-75,)	√	x	√	x	x
	• Improper filtering of request data (CWE-790,791,792, 795,796,797)	√	x	√	x	x
	• Argument injection mechanisms (CWE-88)	√	x	√	x	x

Architecture category	Common vulnerabilities	Insecure API	Unauthorized access	Insecure microservices discovery	Insecure runtime infrastructure	Insecure message broker
Validate input	• XML injection (CWE-91)	√	x	√	x	√
Authorise actors	• Improper handling of privileges and permissions (CWE-266,267, 268,269, 270 271, 272, 273,279,280,281, 732)	√	√	√	√	√
	• Improper access control (CWE-284)	√	√	√	√	√
	• improper or missing authorization mechanisms (CWE-862, 863, 939)	√	√	√	√	√
	• improper management of users (CWE-286)	√	√	√	√	√
	• Bypassing of authorization mechanisms (CWE-566, 639)	√	√	√	√	√
	• Insufficient compartmentalization (CWE-653)	√	√	√	√	√
Authenticate Actor	• Poor password management (CWE-258, 259,262,798,836,916, 640,620, 521)	√	x	x	√	√
	• Bypassing of authentication mechanisms (CWE-288, 289,290, 294,302, 305)	√	x	√	√	√
	• Improper authentication mechanisms (CWE 287, 291, 293,304, 322)	√	x	√	√	√
	• Use of single-factor authentication (CWE 308)	√	x	√	x	√
	• Relying on client-side authentication (CWE-308)	√	x	√	x	√
Encrypt data	• Poor management of credentials information (CWE-256, 257,260)	√	x	√	x	√
	• Inadequate encryption of sensitive data (CWE-312,319, 326,327,328,331,)	√	x	√	√	√
	• Use of hard-coded cryptographic keys (CWE-319)	√	x	x	√	√

Architecture category	Common vulnerabilities	Insecure API	Unauthorized access	Insecure microservices discovery	Insecure runtime infrastructure	Insecure message broker
Encrypt data (continued)	• Use of expire cryptographic keys	√	x	x	√	√
	• Use of weak algorithms (CWE-338, 337,339, 759, 780, 760)	√	x	x	x	√
	• Insecure storage of sensitive information	√	x	√	√	√
Identify Actors	• Improver validation of certificates (CWE-295, 296,298, 299, 370)	√	x	√	√	√
	• Insufficient verification of data authenticity (CWE-345)	√	x	√	√	√
	• Insufficient verification of communication channel request (CWE-940, 941)	√	x	√	√	√
Limit access	• Execution with unnecessary privilege	x	x	√	√	√
	• Improper information exposure (CWE-201, 209, 212)	√	x	√		
Limit Exposure	• Information exposure through error message (CWE-210, 211, 214,550)	√	x	√	√	√
	• Inclusion of untrusted libraries (CWE-829)	√	x	√	√	√
Verify message Integrity	• Lack of proper checksum validation (CWE-353, 354, 649)	√	x	√	√	√
	• Improper exception and error handling (CWE-390, 391, 755)		x	√	√	√
Audit	• Improper output neutralisation for logs (CWE-117)	√	x	x	√	x
	• Omission of security-related information (CWE-223)	√	x	√	√	√
	• Sensitive information exposure through logs (CWE-532)	√	x	x	√	√
	• Insufficient logging (CWE-778)	√	x	x	√	√

(b) A.3.1 Catalog of architecture-level secure coding guidelines

Table A.3.1 is an artefact produced in the preliminary phase of the SAFEMicroservices. The artefact provides a set of guidelines grouped by architecture concept and software engineers can use the artefacts as a quick reference manual to identify which guidelines to apply when dealing with part of the application where the concepts apply.

Table A.3.1 Catalog of architecture-level secure coding guidelines

Architecture category	Common vulnerabilities	Introduction phase	Secure coding guidelines
Validate Input	• Improper input validation (CWE-20)	Design and implementation	• Validate all inputs and validation should consider relevant properties such as length, input type, and acceptable values • Use and specify an output encoding that is supported by a downstream component that consumes its output • Decoded and canonicalized inputs to the microservices current internal representation before validated • Do not use user inputs to constructs all or part of an SQL command without neutralizing special elements • Create a whitelist using regular expressions that define valid input according to the requirements specifications and strictly filter any input that does not match against the whitelist. • Properly encode microservices output and quote any elements that have special meaning between communicating component in the microservices composition.
	• Improper neutralization of request data (CWE-138,150, 643,74,76,77,78,943,95,96, 93)	Implementation	
	• Acceptance of extraneous untrusted data with trusted data	Design and implementation	
	• Cross-site request forgery (CSRF) (CWE-352)	Design	
	• Deserialization of untrusted data (CWE-502)	Design and implementation	
	• Failure to sanitize special elements in request data (CWE-75)	Design and implementation	
	• Improper filtering of request data (CWE-790,791,792, 795,796,797)	Implementation	
	• Argument injection mechanisms (CWE-88)	Design and implementation	
	• XML injection (CWE-91)	Design and implementation	

Architecture category	Common vulnerabilities	Introduction phase	Secure coding guidelines
Authorise actors	• Improper handling of privileges and permissions (CWE-266,267, 268,269, 270 271, 272, 273,279,280,281, 732)	Design and implementation	• Microservices should only use trusted libraries to avoid execution of malicious commands • microservices should run lowest privileges to accomplish task • Ensure appropriate compartmentalization of microservices where trust boundaries are unambiguous • microservices should perform access control validation in the business logic • Ensure that roles are mapped with data and functionality in a microservices • Microservices should use role-based access control to enforce the roles at the appropriate boundaries. • Microservices should not cache sensitive information
	• Improper access control (CWE-284)	Design and implementation, operation	
	• improper or missing authorization mechanisms (CWE-862, 863, 939)	Design and implementation, operation	
	• improper management of users (CWE-286)	Design and implementation, operation	
	• Bypassing of authorization mechanisms (CWE-566, 639)	Design and implementation	
	• Insufficient compartmentalisation (CWE-653)	Design and implementation	
Authenticate Actor	• Poor password management (CWE-258, 259,262,798,836,916, 640,620, 521)	Design and implementation	• Microservices should not hard code credentials in source code or properties files • Password used in the microservices should expire after a given time and be changed • Use vetted authentication frameworks • Microservices should use a multi-factor authentication • microservices should not delegate authentication to clients • Microservices should use strong passwords • Password changes must be verified • Microservices should not use password hash for authentication
	• Bypassing of authentication mechanisms (CWE-288, 289,290, 294,302, 305)	Design and implementation	
	• Improper authentication mechanisms (CWE 287, 291, 293,304, 322)	Design and implementation	
	• Use of single-factor authentication (CWE 308)	Design	
	• Relying on client-side authentication (CWE-308)	Design and implementation	

Architecture category	Common vulnerabilities	Introduction phase	Secure coding guidelines
Encrypt data	• Poor management of credentials information (CWE-256, 257,260)	Design	• Passwords or cryptographic keys should not be stored in property files or hard-coded but consider storing hashes of passwords • Use well vetted algorithm that is considered to be strong • Microservices should not store sensitive information in cookies • do not reuse nonce values • validate for certificate expiry • use approve random number generators that conform to FIPS 140-2
	• Inadequate encryption of sensitive data (CWE-312,319, 326,327,328,331,)	Design and operation	
	• Use of hard-coded cryptographic keys (CWE-319)	Design	
	• Use of expire cryptographic keys	Design	
	• Use of weak algorithms (CWE-338, 337,339, 759, 780, 760)	Implementation	
	• Insecure storage of sensitive information	Design and implementation	
Identify Actors	• Improver validation of certificates (CWE-295, 296,298, 299, 370)	Design and implementation	• Always check that data is encrypted with the intended owner's public key • Validate the certificate full chain trust, host name, expiry date, and revocation status • Verify that the request for communication channel is coming from the expected origin
	• Insufficient verification of data authenticity (CWE-345)	Design and implementation	
	• Insufficient verification of communication channel request (CWE-940, 941)	Design and implementation	
Limit access	• Execution with unnecessary privilege	Design and implementation, configuration, operation	• Use default error messages so that unexpected errors do not disclose sensitive information. • Compartmentalize the microservices into unambiguous trust boundaries and ensure sensitive information does not go over trust boundaries • Perform extensive input validation for any privileged code • Isolate the privileged code as much as possible from other code
	• Improper information exposure (CWE-201, 209, 212)	Implementation	

Architecture category	**Common vulnerabilities**	**Introduction phase**	**Secure coding guidelines**
Limit Exposure	• Information exposure through error message (CWE-210, 211, 214,550)	Design and implementation, operation	• Ensure that microservices does not run in debug mode in production • Ensure that error message are handled internally and do not display error message with sensitive information. • Error message from the runtime environment should not be displayed to the users
	• Inclusion of untrusted libraries (CWE-829)	Implementation	
Verify message Integrity	• Lack of proper checksum validation (CWE-353, 354, 649)	Design and implementation	• Ensure proper implementation of checksums in the protocol design and ensure checksum is added to each message before it is sent • properly check the checksum before parsing the message
	• Improper exception and error handling (CWE-390, 391, 755)	Implementation	
Audit	• Improper output neutralisation for logs (CWE-117)	Design and implementation	• Log all information that may be useful to identify the source and nature of attack. • Protect log files against unauthorized read/write. • Do not deploy microservices in debug mode • do not log excessively
	• Omission of security-related information (CWE-223)	Design and implementation, operation	
	• Sensitive information exposure through logs (CWE-532)	Design and implementation, operation	
	• Insufficient logging (CWE-778)	Operation	

Appendix B

(a) B.1.4 Microservices abuse cases and protection measures

The Table B.1.4 is a catalog of abuse cases and protection measures.

Table B.1.4. Catalog of microservices abuse cases and protection measures

Architecture category	Common vulnerabilities	Abuse or misuse cases	Protection Measures (Including tools and techniques)
Validate Input	• Improper input validation (CWE-20) • Improper neutralization of request data (CWE-138,150, 643,74,76,77,78,943,95,96, 93) • Acceptance of extraneous untrusted data with trusted data • Cross-site request forgery (CSRF) (CWE-352) • Deserialization of untrusted data (CWE-502) • Failure to sanitize special elements in request data (CWE-75,) • Improper filtering of request data (CWE-790,791,792, 795,796,797) • Argument injection mechanisms (CWE-88) • XML injection (CWE-91)	• As an attacker, I can manipulate request parameters to compromise the operation of microservices. • As an attacker, I can supply values as parameters to the API that a microservices implementation uses to determine which class to instantiate and I can then create control flow paths through the microservices that were not intended. • As an attacker, I can manipulate resource identifiers passed on as parameters to microservices API so that I gain control and perform an action on the resource. • As an attacker, I may either alter the path or add/overwrite unexpected parameters in the “query string" on the HTTP query string when calling the microservice REST API. • As an attacker, I may supply multiple HTTP parameters with the same name to cause a microservices to interpret values in unanticipated ways. • As an attacker, I can exploit a microservices composition component by injecting additional, malicious content during its processing of serialized objects.	• Ensure all input content that is delivered to by a microservices is sanitized against an acceptable content specification. • Perform input validation for all content. • Use an input validation framework such as OWASP ESAPI Validation API. • Use static analysis tools such as FindBugs on IDE and continuous integrations toolchains to detect input-validation. • Perform fuzz testing. • Validate object before deserialization process • Limit which class types can be deserialized.

Architecture category	Common vulnerabilities	Introduction phase	Secure coding guidelines
Authorise actors	• Improper handling of privileges and permissions (CWE-266,267, 268,269, 270 271, 272, 273,279,280,281, 732)	• As an attacker I can bypass access restriction and gain access to privileged functionality on microservices. • As an attacker I may elevate my privilege and gain access to the privileged functionality on microservices. • As an attacker I may direct the microservice to execute command on my behalf by loading libraries. • As an attacker I can use a privilege to perform an unsafe action on a microservices. • As an attacker I can use unprotected channels into the microservice.	• Microservices should only use trusted libraries to avoid execution of malicious commands • microservices should run lowest privileges to accomplish task • Ensure appropriate compartmentalization of microservices where trust boundaries is unambiguous • microservices should perform access control validation in the business logic • Ensure that roles are mapped with data and functionality in a microservices • Microservices should use role-based access control to enforce the roles at the appropriate boundaries. Microservices should not cache sensitive information
	• Improper access control (CWE-284)		
	• improper or missing authorization mechanisms (CWE-862, 863, 939)		
	• improper management of users (CWE-286)		
	• Bypassing of authorization mechanisms (CWE-566, 639)		
	• Insufficient compartmentalization (CWE-653)		

Architecture category	**Common vulnerabilities**	**Introduction phase**	**Secure coding guidelines**
Authenticate Actor	• Poor password management (CWE-258, 259,262,798,836,916, 640,620, 521) • Bypassing of authentication mechanisms (CWE-288, 289,290, 294,302, 305) • Improper authentication mechanisms (CWE 287, 291, 293,304, 322) • Use of single-factor authentication (CWE 308) • Relying on client-side authentication (CWE-308)	• As an attacker I can access credential on source code repository to gain access to the system. • As an attacker I can re-use an old password to gain access to the system. • As an attacker I can bypass authentication and gain access to the system. • As an attacker I can use a client session to gain access to microservices.	• Microservices should not hard code credentials in source code or properties files • Password used in the microservices should expire after a given time and be changed • Use vetted authentication frameworks • Microservices should use a multi-factor authentication • microservices should not delegate authentication to clients • Microservices should use strong passwords • Password changes must be verified • Microservices should not use password hash for authentication
Encrypt data	• Poor management of credentials information (CWE-256, 257,260) • Inadequate encryption of sensitive data (CWE-312,319, 326,327,328,331,) • Use of hard-coded cryptographic keys (CWE-319) • Use of expire cryptographic keys • Use of weak algorithms (CWE-338, 337,339, 759, 780, 760)	• As an attacker I can use credentials on file, cookies or source code to access microservices • As an attacker I can access information by listening on the communication channel being used by services • As an attacker I can use expired certificates to access microservices-based • As an attacker I can use brute force to crack encryption algorithms	• Passwords or cryptographic keys should not be stored in property files or hard-coded but consider storing hashes of passwords • Use well vetted algorithm that is considered to be strong • Do not store sensitive information in cookies • do not reuse nonce values • validate for certificate expiry • use approve random number generators that conform to FIPS 140-2

Architecture category	Common vulnerabilities	Introduction phase	Secure coding guidelines
Identify Actors	• Improver validation of certificates (CWE-295, 296,298, 299, 370) • Insufficient verification of data authenticity (CWE-345) • Insufficient verification of communication channel request (CWE-940, 941)	• As an attacker I can use a corrupted certificate to access microservices. • As an attacker I can use another application to gain access to a microservice. • As an attacker I can use an expired certificate to access microservices	• Always check that data is encrypted with the intended owner's public key • Validate the certificate full chain trust, host name, expiry date, and revocation status • Verify that the request for communication channel is coming from the expected origin
Limit access	• Execution with unnecessary privilege • Improper information exposure (CWE-201, 209, 212)	• As an attacker I can inject items to read error messages returned by microservices • As an attacker I can use the privileges of a microservice to gain control of the whole system or environment	• Use default error messages so that unexpected errors do not disclose sensitive information. • Compartmentalize the microservices into unambiguous trust boundaries and ensure sensitive information does not go over trust boundaries • Perform extensive input validation for any privileged code • Isolate the privileged code as much as possible from other code
Limit Exposure	• Information exposure through error message (CWE-210, 211, 214,550) • Inclusion of untrusted libraries (CWE-829)	• As an attacker I can inject items to read error messages returned by microservices • As an attacker I can use known weaknesses in software libraries to gain access to microservices	• Ensure that microservices does not run in debug mode in production • Ensure that error messages are handled internally and do not display error message with sensitive information. • Error message from the runtime environment should not be displayed to the users

Architecture category	Common vulnerabilities	Introduction phase	Secure coding guidelines
Verify message Integrity	• Lack of proper checksum validation (CWE-353, 354, 649)	• As an attacker I can corrupt or alter data or messages in transit	• Ensure proper implementation of checksums in the protocol design and ensure checksum is added to each message before it is sent • Properly check the checksum before parsing the message • Ensure that microservices does not ignore errors
	• Improper exception and error handling (CWE-390, 391, 755)		
Audit	• Improper output neutralisation for logs (CWE-117)	• As an attacker I can read sensitive information on logs • As an attacker I can inject my information on logs • As an attacker I can disguise my actions while manipulating microservices	• Log all information that may be useful to identify the source and nature of attack. • Protect log files against unauthorized read/write. • Do not deploy microservices in • debug mode • do not log excessively
	• Omission of security-related information (CWE-223)		
	• Sensitive information exposure through logs (CWE-532)		
	• Insufficient logging (CWE-778)		

(b) B.2.2 Secure design principles

Table 2.2.2 below provide design principles that software engineers should use when designing secure microservices

Table B.2.2. Microservices secure design principles

	Security designs principles	Principle description	Implementation	Risk to designs principles
1	The principle of least privilege	All components in a microservice composition should be assigned minimum necessary rights when accessing any resource, and the rights should be in effect for the shortest duration necessary.	• Define an unambiguous trust boundary in a microservices composition • Components in a microservices composition should allow sensitive data to go outside the defined trust boundary • Ensure microservice execute with minimum required privileges	CWE-272: Least privilege violation CWE-250: Execution with unnecessary privileges
2	The principle of failing securely	In the event of a component in a microservices composition failing, it should do so securely.	• Ensure redundancy of each component in a microservices compositions • Ensure component in the microservices composition do not propagate sensitive information such as system configuration and user data in the case of exception.	CWE-636: Not failing securely
3	The principle of defense in depth	The components should use layering of security defenses to reduce the chance of a successful attack.	• User layered security mechanisms	CWE-656: Reliance on security through obscurity
4	The principle of economy of mechanism	The components should ensure that multiple conditions are met before granting access permission.	• Avoid complex security mechanisms that cannot easily be understood • Avoid complex data models • Adopt secure programming principles • Microservices must avoid having multiple subjects sharing access mechanisms	CWE-637: Unnecessary complexity in the protection mechanism

	Security designs principles	Principle description	Implementation	Risk to designs principles
5	The principle of separation of privilege	The design of each component should be kept simple.	• Ensure that multiple conditions are met before permitting access to a microservices composition resource. • Use isolated accounts with limited privileges to use for a single task	CWE-269: Improper privilege management CWE-268: Privilege chaining
6	The principle of open design	The security of microservices composition should not be based on the secrecy of its design or implementation	• Use publicly vetted algorithms and procedures that have undergone more extensive security analysis and testing	CWE-656: Reliance on Security Through Obscurity
7	Principle of complete mediation	Every access to every resource must be validated for authorization.	• Invalidate cached privileges whenever there is update of user privileges. • Do not access control decisions as much as possible.	CWE-638: Not Using Complete Mediation

(c) B.2.3 Security standards

This artefact is produced by the SAFEMicroservices planning phase. The artefact provides a list of standards that software engineer should follow when designing and deploying microservices.

Table B.2.3. Microservices security standards

	Microservices security design and deployment standards	Architecture components
1	Any communication with a microservice must be done via API Gateway to provide load balancing, and a standard set of security capabilities and communication to API gateway should be authenticated	Microservices
2	Each microservice must be protected using a defense in depth approach	Microservices
3	The microservices composition must use a well-known and secure open standard protocol for centralized authentication using tokens. The token must be generated using an algorithm that follows the cryptography standard and should have an associated time to live	Microservices

	Microservices security design and deployment standards	Architecture components
4	Authentication Tokens must be encrypted	Microservices
5	Each microservices must have a unique API key for calling another microservice	Microservices
6	API calls made by users and systems must be limited to only those necessary for those users or systems to perform their functions	Microservices
7	All API requests must be logged to a centralized logging and monitoring system	API gateway
8	A tool to monitor and visualize inter-microservice communication must be deployed as part of the management capabilities of the microservices architecture	Microservices
9	All communication in the microservices composition must use Transport layer security	Microservices, API gateway, services registry, message broker
10	All microservices composition components must run in an approved application container technology	Microservices, API gateway, services registry, message broker
11	Containers must only provide capabilities required to support the microservices running it, and nothing more	Runtime infrastructure
12	Container should not run network specific operations but network configuration should only be applied to the container at startup and not be dynamically assigned or modified	Runtime infrastructure
13	All the code or libraries required to execute within the container must be within the container image and never be loaded dynamically	Runtime infrastructure
14	Container should run with minimum set of privileges required to perform its function	Runtime infrastructure
15	Inter-container communication should only be done via port binding, with ports explicitly opened in a container configuration file	Runtime infrastructure
16	The file system in containers should set to be read only to prevent malicious overwrites	Runtime infrastructure
17	Containers hosting microservices should only expose a single port or the minimal number of ports required to support the microservices	Runtime infrastructure
12	Connectivity to microservices should be controlled through IP Filtering technologies	Runtime infrastructure
13	Operation of microservices, their resource consumption and performance should be monitored and spikes in consumption addressed through capacity management activities	Runtime infrastructure

(d) B.2.4 Monitoring guides

Table E.2.3 Provide monitoring guidelines that software engineers should use when design the monitoring components of microservices

Table B.2.4. Monitoring guidelines

	Microservices monitoring guidelines	**CWE Vulnerabilities**
1	Log all information important for identifying the source or nature of an attack	CWE-223: Omission of Security-relevant Information
2	Do not log sensitivity information on the log files	CWE-532: Information Exposure Through Log Files
3	Log information on user events in much details so that attack behavior can be detected and ensure that all login successes and failures are logged.	CWE-778: Insufficient Logging
4	Do not log the user input data into log files without neutralizing the input	CWE:117 Improper output neutralization for logs
5	Do not log unnecessary information that makes it hard to process log files or perform a forensic analysis in the event of attack	CWE: 779: Logging of excessive data
6	Use a centralized logging approach that supports multiple levels of logging details	CWE: 779: Logging of excessive data CWE-778: Insufficient Logging
7	Always make sure that the level of logging is set appropriately in a production environment	CWE-778: Insufficient Logging
8	Make sure the log level is not set to debug mode debug log files before deploying the application into production.	CWE-532: Information Exposure Through Log Files
9	The log files should be protected against unauthorized read/write.	CWE-532: Information Exposure Through Log Files

Appendix C

(a) C.1. Mapping security threats to CAPEC attack mechanisms

Table C.1. shows the list of attack mechanisms that can be used to exploit weakness in microservices. The table is meant to guide software engineers understand the attacks associated with each threat

Table C.1 Mapping security threats to attack patterns

Security threats	**Applicable CAPEC mechanisms of attack**
Insecure application programming interfaces	• CAPEC-152: Inject unexpected items • CAPEC-210: Abuse existing functionality • CAPEC-255-Manipulate data structures • CAPEC-223: Employ probabilistic techniques • CAPEC-118: Collect and analyze information • CAPEC-225: Subvert access control • CAPEC-156: Engage in deceptive interaction
Unauthorized access,	• CAPEC-225: Subvert access control
Insecure service discovery	• CAPEC-152: Inject unexpected items • CAPEC-210: Abuse existing functionality • CAPEC-255-Manipulate data structures • CAPEC-223: Employ probabilistic techniques • CAPEC-118: Collect and analyze information • CAPEC-156: Engage in deceptive interaction
Insecure message broker	• CAPEC-210: Abuse existing functionality • CAPEC-255-Manipulate data structures • CAPEC-225: Subvert access control • CAPEC-156: Engage in deceptive interaction • CAPEC-118: Collect and analyze information
Insecure runtime infrastructure	• CAPEC-225: Subvert access control