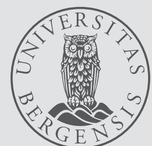
Exploring Microservice Security

Tetiana Yarygina

Thesis for the Degree of Philosophiae Doctor (PhD)
University of Bergen, Norway
2018

UNIVERSITY OF BERGEN



Exploring Microservice Security

Tetiana Yarygina

Thesis for the Degree of Philosophiae Doctor (PhD)
at the University of Bergen

2018

Date of defence: 01.10.2018

© Copyright Tetiana Yarygina

The material in this publication is covered by the provisions of the Copyright Act.

Year: 2018

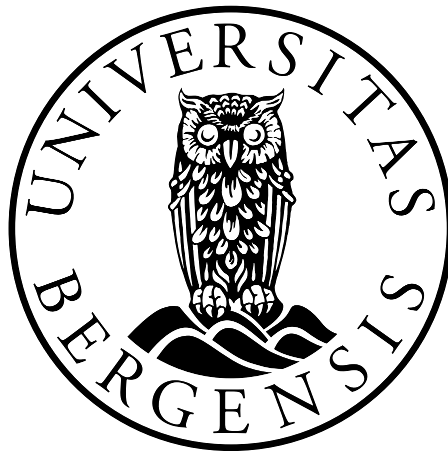
Title: Exploring Microservice Security

Name: Tetiana Yarygina

Print: Skipnes Kommunikasjon / University of Bergen

Exploring Microservice Security

Tetiana Yarygina



Dissertation for the degree of Philosophiae Doctor (PhD)

Department of Informatics
University of Bergen

July 4, 2018

ISBN 978-xx-xx-xxxx-x
University of Bergen, Norway. Submitted 2018-07-04, final print 2018-08-xx.
Papers reprinted by permission from Springer and IEEE. All rights reserved.
All other text, illustrations and photos © 2018 Tetiana Yarygina.
Printed by Skipnes Kommunikasjon.

Prepared with LaTeX and set in *EB Garamond*, Universalis ADF Std, HESSE ANTIQUA and `txtt`.

Acknowledgements

I thank my PhD advisors, Assoc. Prof. Anya Helene Bagge and Prof. Jaakko Järvi, for their extensive knowledge, insightful suggestions, good judgment and support throughout the work on this thesis. I express my gratitude to Prof. Kjell Jørgen Hole and Prof. Øyvind Ytrehus for their valuable guidance and encouragement on the early stages of the research work.

I am grateful to my PhD opponents, Dr. Tor Erling Bjørstad and Prof. Coen De Roover, for their time and comments.

I would like to thank my colleagues and friends at UiB, Christian Otterstad, Anna Maria Eilertsen, Stian Fauskanger and May-Lill Bagge, for the stimulating discussions, useful feedback and motivation. I am truly grateful for the time we worked together.

I especially thank my parents and my brother for their constant support and unconditional love through this journey. I also thank my childhood friend, Iryna, for her encouragement and friendship that I needed during these years.

Abstract

Due to rapid transitioning towards digitalized society and extended reliance on interconnected digital systems, computer security is a field of growing importance. Software that we build should be secure, resilient and reliable both against accidents and targeted attacks.

The microservice architecture, or concisely *microservices*, is a recent trend in software engineering and system design. Microservices are a way to build scalable and flexible distributed applications as a collection of loosely coupled services communicating over a network.

In this thesis, we study the microservice architectural style from a security perspective. The contributions are as follows. We show that microservice architecture has inherent security benefits in terms of isolation and diversity. We explore how these inherent security benefits of microservices can be improved even further by maximizing interface security, avoiding unnecessary node relationships, introducing asymmetric node strength, and using N-version programming.

We design a taxonomy of microservice security giving an overview of the existing security threats and mitigations. In this thesis, we argue that the defense in depth principle should be adopted for microservices. We discuss several prominent microservice security trends in industry. Furthermore, we present an open source prototype security framework for microservices.

We take the defense in depth principle even further by focusing our attention on the self-protection and adaptive security properties. Also, we propose an architecture of an automated intrusion response system for microservices that uses game-theoretic approach. Finally, we analyze the security properties of the REST style, the most typical microservice integration solution.

List of papers

1. Tetiana Yarygina, Anya Helene Bagge, *Overcoming Security Challenges in Microservice Architectures*, In: 12th IEEE Symposium on Service-Oriented System Engineering. SOSE 2018. pp.11-20. DOI 10.1109/SOSE.2018.00011.

2. Christian Otterstad, Tetiana Yarygina, *Low-Level Exploitation Mitigation by Diverse Microservices*, In: De Paoli F., Schulte S., Broch Johnsen E. (eds) 6th European Conference on Service-Oriented and Cloud Computing. ESOCC 2017. Lecture Notes in Computer Science, vol 10465, pp.49-56. Springer, Cham, DOI: 10.1007/978-3-319-67262-5_4.

3. Tetiana Yarygina, Christian Otterstad, *A Game of Microservices: Automated Intrusion Response*, In: Bonomi S. and Rivière E. (eds) 18th IFIP International Conference on Distributed Applications and Interoperable Systems. DAIS 2018. Lecture Notes in Computer Science, vol 10853, pp.1–9. Springer. DOI: 10.1007/978-3-319-93767-0_12.

4. Tetiana Yarygina, *RESTful Is Not Secure*, In: Batten L., Kim D., Zhang X., Li G. (eds) 8th International Conference on Applications and Techniques in Information Security. ATIS 2017. Communications in Computer and Information Science, vol 719, pp.141-153. Springer, Singapore. DOI: 10.1007/978-981-10-5421-1_12.

Contents

Acknowledgements iii

Abstract v

List of papers vii

1 Introduction 15
- 1.1 Motivation 15
- 1.2 Research Questions 16
- 1.3 Outline 17

2 Background 19
- 2.1 Microservices and Evolution of Distributed Systems 19
  - 2.1.1 Definition of Microservices 19
  - 2.1.2 The Underlying Principles of Microservices 20
  - 2.1.3 The Promise and Limitation of Microservices 22
  - 2.1.4 Service-Oriented Architecture 25
  - 2.1.5 Future of Microservices 26
- 2.2 Security Primer 27
  - 2.2.1 The Core Security Concepts 27
  - 2.2.2 Public Key Infrastructure 28
  - 2.2.3 Delegated Authorization and Authentication 30
  - 2.2.4 SOA Security 33
  - 2.2.5 Security Measures 34
- 2.3 On Microservice Integration Styles 36
  - 2.3.1 What is REST 37
  - 2.3.2 Loose Coupling 38
  - 2.3.3 How Dumb Are the Pipes? 40
  - 2.3.4 Security, Performance and Engineering Cost 41
  - 2.3.5 Discussion 42
  - 2.3.6 Conclusion 43

3 Summary of Papers 45

4 Paper I: Overcoming Security Challenges in Microservice Architectures 51
4.1 Introduction . . . 53
4.2 What Microservices Really Are . . . 55
4.2.1 Defining Microservices . . . 55
4.2.2 Defining Service-Oriented Architecture (SOA) . . . 56
4.2.3 Déjà Vu . . . 57
4.2.4 Distributed Systems . . . 57
4.2.5 Microservices in Context of Other Technologies . . . 58
4.2.6 Summary: Essence of Microservices . . . 58
4.3 Layered Security for Microservices . . . 59
4.3.1 Taxonomy of Microservice Security . . . 59
4.3.2 Attack Model: Redefining Perimeter Security . . . 61
4.4 Security Implications of Microservice Design . . . 61
4.5 Emerging Security Practices . . . 63
4.5.1 Mutual Authentication of Services Using MTLS . . . 63
4.5.2 Principal Propagation via Security Tokens . . . 64
4.5.3 Fine-Grained Authorization . . . 66
4.6 Microservice Security Framework . . . 67
4.6.1 Design and Implementation . . . 67
4.6.2 Experiment . . . 68
4.6.3 Evaluation . . . 71
4.7 Conclusion . . . 71

5 Paper II: Low-level Exploitation Mitigation by Diverse Microservices 77
5.1 Introduction . . . 79
5.2 Model Overview and Exploitation Analysis . . . 80
5.2.1 Model Overview . . . 81
5.2.2 Exploitation Overview . . . 81
5.3 Microservice Architecture and Its Security Merits . . . 82
5.3.1 Microservice Design Patterns Affecting Security . . . 82
5.3.2 Security Considerations . . . 83
5.3.3 Security Through Diversity . . . 85
5.4 The Security Monitor Service . . . 86
5.4.1 Design Overview . . . 86
5.4.2 Evaluating the Security Monitor . . . 88
5.5 Proof of Concept by Example . . . 89
5.5.1 System Architecture . . . 89
5.5.2 Exploitation Example . . . 90
5.6 Conclusion . . . 92

6 Paper III: A Game of Microservices: Automated Intrusion Response 95
6.1 Introduction . . . 97
6.2 Security Games: Assumptions and Solutions . . . 98

6.2.1 Finite Dynamic Two-player Zero-sum Game 99
6.2.2 Minimax 99
6.3 Proposed Architecture 100
6.3.1 Network Mapping 100
6.3.2 Intrusion Detection and Events Reporting 101
6.3.3 Event Evaluation Function 101
6.3.4 Decision Function 101
6.3.5 Intrusion Response: Defender's Actions 102
6.4 Evaluation and Discussion 103
6.5 Conclusions 104

7 Paper IV: RESTful Is Not Secure 107
7.1 Introduction 109
7.2 Overview of Security Mechanisms for the Modern Web 110
7.2.1 Token-based Authentication 111
7.2.2 Client Side Request Signing 112
7.2.3 Delegated Authorization and Shared Authentication . . 113
7.3 REST Architectural Style and Security 113
7.3.1 Not Designed with Security in Mind 114
7.3.2 Stateless Constraint 114
7.3.3 Other Constraints Affecting Security 117
7.4 The Way Forward 118
7.4.1 Security Failure of REST 118
7.4.2 What to Do 118
7.4.3 Future Research 119

8 Conclusion and Future Work 123

List of Figures

1.1 Transitioning to microservices. The granularity of components increases from left to right. 16

2.1 Virtual machines versus containers. 22
2.2 OAuth 2.0 Implicit Flow. 31
2.3 OAuth 2.0 Authorization Code Flow. 31
2.4 The sophistication of attack tools versus required attack knowledge. 36
2.5 A relative complexity scale of the existing service integration solutions. 41
2.6 A microservice-based bank built using multiple inter-service communication styles. 43

4.1 Microservice architecture in perspective. 56
4.2 Microservices redefine perimeter security and move towards defense in depth. 61
4.3 A generic solution for microservice trust based on MTLS. 65
4.4 A generic token-based authentication scheme for microservices. . 66
4.5 Experiment setup: payment operation. 69
4.6 Performance of the bank model under load of 50 test clients making payments. 70

5.1 Attacking a microservice architecture with diverse microservices running in virtualized environments on networked machines. . . 82
5.2 Depiction of an unnecessary edge, exposing additional attack surface. 84
5.3 The use of asymmetric node strength to defend against low-level attacks. 85
5.4 A security monitor dealing with an infection in an N-version system. 87

6.1 Security game between the defender and attacker. 99
6.2 An overview of the proposed monitor architecture. 100

7.1 The hierarchical relation between common security mechanisms. . 111
7.2 Making the right security decision. 119

1

Introduction

1.1 Motivation

Computerization undeniably impacts our lives and shapes society. Personal computers are ubiquitous: desktops, laptops, tablets, smart-phones, smart-watches, and various smart-home items. Implantable and wearable medical devices such as heart pacemakers and insulin pumps are increasingly commonplace. Autonomous cars promise to make the roads safer and more efficiently utilized. DNA sequencing is broadening the biological and medical knowledge. Artificial intelligence promises to supply us with the most relevant targeted advertisement and friendly chat-bots, among other things. All these advances and many more to come are possible because of the vast affordable computational resources.

The transitioning towards a digitalized society brings great benefits and great challenges. Digitalization of public services, such as electronic voting, taxation, medical records, population census, and many more, aims to make public services more efficient, transparent, and accessible. Cashless society and growing digital economy are global trends supported by the rise of mobile payments and digital currencies. Similarly to how computers are pervasively interconnected via World Wide Web, people are now connected through various digital social networks. The importance of digital life and digital presence will only increase.

Due to extended reliance on interconnected digital systems, digital crime is a serious problem that threatens financial, medical and government systems, industrial equipment, automobile, and aviation industries, as well as private individuals. The impact of digital crime can be enormous, ranging from privacy violation and sensitive data taken for ransom (WannaCry attack) to threats on national nuclear power programs (Stuxnet attack). Cyberwarfare can be viewed as a modern form of warfare as more and more countries establish national cybersecurity forces.

In the age of digital crime, computer security (cybersecurity) is of critical importance. The way we build software has serious security implications because most of the cyber attacks exploit existing vulnerabilities in software. Therefore, it is essential to understand how the common software architectural styles address security and how they can be improved.

Microservice architecture, or concisely microservices, is a recent trend in software engineering and system design. Microservices are a way to build an application as a collection of loosely coupled services communicating over a network boundary.

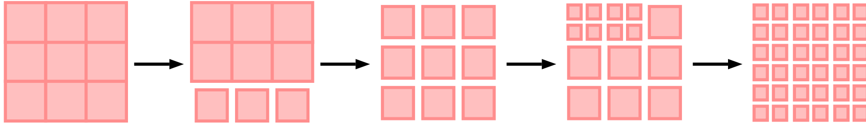


FIGURE 1.1: *Transitioning to microservices. The granularity of components increases from left to right.*

The microservice architecture enforces modular structure and strict separation of concerns that allows creating highly scalable and flexible distributed systems. The process of gradually transitioning from a modular non-distributed system towards microservices is shown in Figure 1.1.

A survey conducted by NGINX [86] in November 2015 showed that 33% of IT companies had microservices in production, and even more were planning to start using microservices. In April 2018, a global survey [65] of IT specialists within a diverse set of industries including technology, finance, healthcare, and others, found that 60% of respondents had microservices in pilot or production. Moreover, 86% of respondents [65] believed microservices to be *the default* architecture by 2023. The microservice architecture market is proliferating and expected to reach 32 billion US dollars by 2023 [52].

1.2 Research Questions

This thesis analyses microservice architecture from a security perspective. Since microservice security is an emerging research area, there is a lack of dedicated scientific literature to be surveyed. Microservice security is mostly discussed by practitioners in blogs, online articles, and development conference talks. However, such sources take a practical engineering perspective on the subject and are limited in detail and scope. The holistic view on microservice security is missing. Therefore, the research questions formulated for this thesis are broad in scope.

The following three research questions about the important aspects of microservice security were chosen.

- **RQ1** a) Do microservices have security concerns distinct from those of SOA and distributed systems? b) If so, what is the overlap between microservices and SOA/distributed systems security? What can be reused?

- **RQ2** What are the security challenges on the path to microservice adoption? How can these challenges be addressed?

- **RQ3** Can the microservice architectural style lead to better security? If so, what are the opportunities enabled by it?

This study was exploratory and interpretative in nature. The methodological approach taken in this study is a mixed methodology based on designing, prototyping, and evaluating the applicable security solutions. Primarily qualitative methods

were used in this investigation since no standard quantifiable security metrics exist: security is difficult to measure.

1.3 Outline

Chapter 2 provides the relevant background information and is organized as follows. Section 2.1 introduces microservice architecture, explains the core microservice principles, discusses the transition from SOA to microservices, future of microservices, and current challenges in microservice adoption. Section 2.2 gives an overview of the important security primitives, principles, and standards that are relevant for microservices. Section 2.3 examines the popular microservice integration styles and inter-service communication options, including the REST architectural style.

Chapter 3 summarizes the contributions of the four research papers that comprise this thesis. The four research papers are presented in Chapters 4, 5, 6, and 7. Chapter 8 concludes the thesis by reflecting on the aforementioned research questions and providing directions for future research.

2 Background

2.1 Microservices and Evolution of Distributed Systems

This section explains microservice architecture through the underlying principles of separation of concerns, continuous delivery, and virtualization. We discuss the advantages and disadvantages of the architecture. Improved scalability, reduced development time, and technology heterogeneity are identified as the main benefits of microservices. The drawbacks are similar to those of distributed systems and lie in the areas of fault tolerance, software testing, distributed transactions and data consistency, and infrastructure complexity. This section explains the transition from Service-Oriented Architecture (SOA) to microservices and concludes by discussing the future of microservices.

2.1.1 Definition of Microservices

The term "microservices" is not strictly defined, although a variety of definitions exist. All these definitions involve a notion of a service, which we define as follows in this thesis:

Definition 1. *A service is a self-contained unit of business functionality that can be accessed remotely and may consist of other underlying services. Communication between services occurs through network calls rather than system calls.*

The two often cited definitions of microservices are listed below. Newman [85] defines microservices as small autonomous services built around the following seven principles: model [services] around business concepts, adopt a culture of automation, hide internal implementation details, decentralize all things, isolate failure, and make services independently deployable and highly observable.

Lewis and Fowler [38] view microservices as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

Definition 2. *In this thesis, we define microservice architecture as a specialized variant of service-oriented architecture that emphasizes fine-grained separation of concerns, continuous delivery, and virtualization.*

The reasoning behind choosing this definition is given below.

2.1.2 The Underlying Principles of Microservices

There are several pillars that microservice architectures build upon. These technologies and principles enable microservice architectures; without them, microservices would not come into existence.

Separation of Concerns

Often, an effective way to solve a complex problem is to decompose it into smaller and more manageable parts. Separation of concerns is a basic software engineering principle [62, p.85] that enforces such an approach. Separation of concerns is achieved through encapsulation (information hiding) such that logically related elements of the system are grouped together and their implementation details are hidden behind a well-defined interface.

A program built after the strict separation-of-concerns principle is a *modular program*. The notion of software modularity dates back to 1969 [72]. The Unix operating system is a prominent example of modular design. When concerns are separated into layers, the term layered architecture is used. In object-oriented programming, concerns are separated into objects. In Service-Oriented Architecture (SOA), concerns are separated into services.

Benefits of separation of concerns (and modularity) are many: decreased complexity, improved comprehension, simplified development and maintenance process. Individual concerns (or modules) can be developed in parallel and modified independently because a change in one component should have no or minimal effect on the others. Furthermore, no knowledge of internal implementation details of other components except the one to be modified is needed.

The primary objectives of modularity, *low coupling* and *high cohesion*, are discussed in Section 2.3.2. Low coupling and high cohesion seek to fulfill the goals of flexibility, scalability, and fault tolerance in software.

Continuous Delivery and DevOps

Continuous delivery and DevOps lie at the core of microservices. The IT industry is a dynamic environment where responsiveness to change is of critical importance. An ability to release code changes fast can be a deciding factor for business success. Methods and tools that accelerate the delivery of changes are desired.

A survey of deployment practices in the industry by Leppänen et al. [64] showed that the time to market could be as low as 20 minutes (a speed with which a development team can push a change to production using its *normal* development workflow). Time to market in order of minutes is a significant speedup compared to the still existing practice of having software releases as infrequently as a few times per year.

DevOps is a software engineering practice that emphasizes the importance of collaboration between development (Dev) and IT operation (Ops) teams to shorten

development cycles and increase the frequency and dependability of software releases. While DevOps focuses on the organizational side of the problem, continuous integration and continuous delivery approaches are concerned with the technical aspects of automation.

Continuous integration is the process of automated software building and testing whenever changes are detected in a shared version control repository. *Continuous deployment* takes this process even further by automatically deploying any new successful build to a production environment. *Infrastructure automation* is an important component of continuous deployment that allows applying the same configuration to any number of nodes. Chef[1] and Puppet[2] are examples of popular open source tools for infrastructure automation. Complete transitioning to continuous deployment and full automation is nontrivial.

Cloud, Virtualization and Containerization

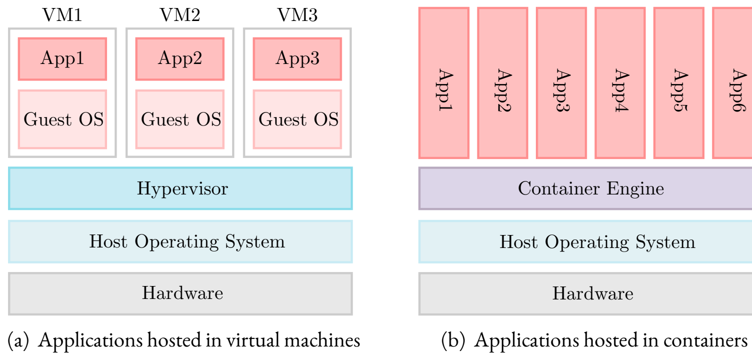
The last but not least group of technologies that enable microservices is a combination of cloud, virtualization, and containerization. The cost of computer hardware, such as CPU, memory, and storage, has been steadily decreasing, while the performance has been improving. Together with the availability of high-throughput networks, these factors made the idea of utility computing (a type of on-demand computing) a reality again.

Platform virtualization is a technology that allows one physical server to run multiple virtual machines, i.e., emulations of a computer system. The software that creates a virtual machine on the actual hardware and abstracts the machine's resources is called a *hypervisor*. The main benefits of virtualization are improved hardware-resource utilization, live migration, snapshots, and failover. The performance overhead limits the use of virtualization in certain cases, such as time-critical applications with constant loads.

Virtualization enables cloud computing. *Cloud computing* is a paradigm that allows different parties to access a shared pool of automatically provisioned and usually virtualized system resources. The term was popularized in 2006 with the launch of Amazon Elastic Compute Cloud (EC2). The three standard models of cloud computing (in order of increasing abstraction) are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The primary deployment methods are a private, public, and hybrid cloud.

Containerization is another trending technology. *Containerization* is a way to encapsulate an application and its dependencies into a self-contained portable unit called a *container*. While a virtual machine emulates a whole environment with its own operating system, the containers all use the same host operating system, as shown in Figure 2.1, and, therefore, are more lightweight. Containerization can be viewed as virtualization on the level of operating system. Both virtual machines and containers share the same goal of portability: the software delivered in a virtual

[1] https://www.chef.io/chef/
[2] https://puppet.com/



(a) Applications hosted in virtual machines (b) Applications hosted in containers

FIGURE 2.1: *Virtual machines versus containers.*

machine or a container will execute identically on any supported platform without additional configuration or installation effort.

Although containers are an old concept dating back to 1998 (the jail utility in FreeBSD), the release of Docker, an open source containerization tool, in 2013 made them mainstream. Containers and microservices fit well together. Docker containers are the default choice for microservice deployment nowadays [81].

2.1.3 The Promise and Limitation of Microservices

Advantages

The main benefits of microservices can be grouped into the following categories:

- *Scalability.* There are three main techniques for scaling [125, p.12]: hiding communication latencies (i.e., using asynchronous communication and data cashing), distribution, and replication. Microservices utilize distribution and replication fully. Furthermore, the microservice architecture allows for optimized selective scaling such that only the types of microservices that are in demand at the moment are scaled up whereas the ones that are underused can be scaled down. The fact that system load is non-uniform is at the heart of microservice architectures.

  In a modular architecture, different modules are likely to have different system resource requirements: while some modules are CPU-intensive, others can be memory-intensive. A collective deployment of such modules requires overall more powerful hardware than if deployed separately in specialized instances in a cloud environment. Microservice architectures facilitate the latter. Microservices can be scaled independently, which leads to better system resource utilization.

- *Development time.* Despite the need for a supporting infrastructure, *individual* microservices are easy to develop. Microservice design improves compre-

hension by decreasing *individual* service complexity. Microservices can be developed and deployed independently which reduces the coordination overhead between different developers or teams.

In general, computational resources become cheaper while developers' time does not. This makes the trade-off between time to market and performance optimization more prominent. In the microservice architectures, developers' time is a priority while the performance tends to be lower than in non-distributed applications. Microservices emphasize automation to optimize development, deployment, and maintenance efforts.

- *Technology heterogeneity.* The technological landscape changes rapidly: concepts, tools, frameworks, and programming languages are abandoned, while new ones appear and gain popularity fast. Although homogeneous solutions were a common goal in the past, homogeneity does not scale well. For systems beyond a certain size, even if maintaining homogeneity is possible, the price of it increases dramatically.

  Microservice architecture promotes technology heterogeneity and diversity. Individual microservices can be implemented in different programming languages and use different frameworks. Various versions of the same microservice can co-exist given that the interfaces stay unchanged.

Microservice benefits are explicitly derived from the benefits of the underlying principles and technologies discussed earlier. Many of the microservice benefits are the same as the benefits of distributed computing.

Disadvantages

> *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*
>
> Leslie Lamport, 1987

Distributed systems are inherently more complex than non-distributed systems. All the drawbacks of distributed systems follow microservices. However, techniques and solutions to counteract these drawbacks are emerging.

- *Fault tolerance.* Software fault tolerance is an ability of a computer system to continue to operate acceptably despite the presence of partial failures. Software fault tolerance is a valuable property and a design goal of distributed systems [125, p.321]. A process where a failure of one component triggers the failure in other components is called a *cascading failure*. Cascading failures in distributed systems can have catastrophic consequences by making the entire system non-functional.

To prevent cascading failures in the microservice world, a design pattern called *circuit breaker* [80, 85] is often employed. Similarly to an electrical circuit breaker, a software circuit breaker limits the impact of individual service failures from cascading to other services. A circuit breaker can be seen as a wrapper or proxy around a service that adjusts the service network behavior. If there are too many unsuccessful connections made to a particular service, a circuit breaker temporarily throttles attempts at further connections. To handle unresponsive services, a circuit breaker will enforce connection timeouts. Hystrix latency and fault tolerance library by Netflix[3] is a popular implementation of the circuit breaker pattern for the Java programming language.

- *Software testing.* Testing large-scale distributed systems is nontrivial. The known cases of prolonged downtime due to cascading failures in microservice-based systems show the importance of systematic resilience testing [46]. Traditionally, tests are run in a test environment before releasing software into production. *Chaos engineering* [7] is an emerging discipline concerned with systematic resilience testing of software systems *in production* environments.

  The idea of artificially injecting failures into the production environment and continuously stress-testing the system is at the core of chaos engineering. The scale of such intentional failures may vary significantly. Examples of such failures [7] are the termination of virtual-machine instances, latency injection into requests between services, failing requests between services, failing an internal service, and making a big part of the service infrastructure unavailable.

- *Distributed transactions and data consistency.* It can be challenging to perform changes that affect multiple services, such as interface changes and distributed transactions. In the microservice architecture, each microservice own its data. There is no complex database shared between all of the services. Instead, the data is partitioned into smaller databases that are owned by relevant microservices. A need to update multiple databases that belong to different services is likely to arise, leading to data consistency issues.

  ACID (Atomicity, Consistency, Isolation, Durability) properties in a distributed transaction are difficult to achieve. Solutions such as compensation over two-step-commit [125, p.355] allow to apply several distinct changes as a single operation (atomic operation) and rollback if any of the changes are unsuccessful. *Eventual consistency* is a consistency model with weaker constraints than continuous consistency. Eventual consistency implies that over time all the replicas converge toward identical copies of each other via updates that are guaranteed to propagate [125, p.289].

- *Infrastructure complexity.* Microservice architecture depends heavily on infrastructure automation. Service discovery and service management func-

[3] https://github.com/Netflix/Hystrix

tionality are necessary infrastructure components. Furthermore, monitoring tools are a required when operating a large microservice system.

The fact that microservices gained wide adoption despite the aforementioned drawbacks implies that the benefits of microservices outweigh the disadvantages in practice. Some of the microservice advantages and disadvantages are also applicable to SOA.

2.1.4 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is a close relative of microservices. The first report about SOA [114] was published in 1998. In his book [57], Josuttis defines SOA as "an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners." However, similarly to microservices, no consensus exists on the precise definition of SOA.

The core technical concepts of SOA [57] are services, interoperability, and loose coupling. The notions of services and loose coupling were introduced earlier in the microservice context. Interoperability, as the ability of computer systems or software to interact and exchange information, can be achieved in different ways. A typical implementation pattern for SOA is Enterprise Service Bus (ESB) [57] which enables service consumers to call the service providers via an intermediary.

SOA is commonly implemented using web services specifications collectively referred to as WS-* stack. Basic standards, such as Simple Object Access Protocol (SOAP) [42] intended for exchanging structured information in a distributed environment, lie at the core of SOA. Multiple additional standards, such as WS-Security and WS-Notification, extend the basic framework by addressing non-functional requirements, e.g. security. Yet, there is no single set of web services specifications. Many different specifications were developed by various entities over the years resulting in practitioners' confusion, misunderstanding, and a loss of trust.

It should be mentioned that web services are only one possible way of implementing SOA. Other technologies such as message queues and remote procedure calls (RPC) can be used.

SOA evolved as a solution to the desire for flexibility, scalability, and fault tolerance in large distributed systems. Such systems are usually complex, heterogeneous, and contain legacy components as a consequence of a long lifetime. However, transitioning to SOA is nontrivial. For Credit Suisse, a global financial institution with thousands of in-house developers, it took around ten years to adopt the service architecture and split the legacy system into more than 1000 services [82].

Despite its popularity, the understanding of how to do SOA right is still lacking. Many problems faced by SOA practitioners rest with underlying technologies, such as the WS-* stack and vendor middleware (ESB), or design misconceptions due to a lack of guidelines and best practices for implementing SOA in the real world.

From SOA to Microservices

The distinction between SOA and microservices is a debated question [138]. On a conceptual level, SOA and microservices are identical: they try to solve the same problems of system flexibility and scalability by using similar principles of modularity and distribution. However, when looking closer, differences start to appear.

Richardson [107, p.8] views microservices as SOA without the commercialization, WS-* stack, and ESB. Newmann [85, p.8] views microservices as a specific approach to SOA. This view is further supported by Zimmerman [138], who defines microservices as a particular implementation and deployment variant of SOA.

The pillars that microservice architecture relies on, fine-grained separation of concerns, continuous delivery, and virtualization, as well as all its supporting technologies, have seen rapid development in the recent years. Continuous delivery and virtualization solutions were not readily available for a mass-scale use when SOA emerged. Microservices are reaching a height of adoption unseen by SOA. Both SOA and microservices constitute sequential steps in a gradual development and evolution of distributed systems.

2.1.5 Future of Microservices

Big Picture

Not every system needs to be distributed. Moreover, not all distributed systems should be microservice-based. Neither SOA nor microservices are a free lunch or a silver bullet [57, 85]. Both advantages and disadvantages of microservices should be acknowledged and weighted against one another when deciding on an architectural style for a system. In some cases, such as when performing time-critical operations, the performance overhead introduced by microservices can be unacceptable and pointlessly expensive.

Microservices are often contrasted with monolithic applications. Such comparisons are often oversimplified and overlook the fact that monolithic systems, i.e., non-distributed systems, can have a modular design and focus on frequent releases and automation. While microservice architecture naturally enforces these properties, it is not the only way to achieve them.

Challenges

Since microservices are still a quite new architecture, there are many challenges on the path to microservice adoption. For example, it is unclear what is the best way to decompose existing systems into microservices and how to choose an appropriate service granularity level. Research on the tools that facilitate decomposition, e.g. Service cutter [43], started to appear. Moreover, the topic of microservice security received undeservedly little attention both in industry and academia.

Despite active development, there is no theoretical framework to follow when building microservice applications. Similarly to the SOA case, the lack of clear guidelines may complicate migration to microservices. Microservice architecture leaves many decisions to the discretion of the developers. Moreover, a definition of an ar-

chitectural style usually involves a set of constraints. Absence of such constraints for microservices makes the whole concept less precise.

Trends

Change is the only certainty for any area of software engineering. Currently, microservices are trending. Similarly to how SOA popularity is declining, in the future microservices are also likely to be replaced with a different technology that will solve the existing and new problems even better. The timeframe for the change, however, is difficult to predict.

Automation is a driving force of microservices and a critical factor for microservice adoption. More tools, frameworks, and platforms are likely to appear to support even higher degrees of automation in all aspects of the microservices lifecycle. It is also likely that we will see an active development in the areas of microservice security, decomposition, monitoring, testing, and orchestration in the future.

Serverless computing and the *function-as-a-service model* are the latest developments in cloud computing that go hand in hand with microservices. Serverless computing is a type of utility computing where infrastructure provisioning is fully automated and the resources consumed by functions execution are measured with high precision. AWS Lambda, the first serverless computing platform that was launched by Amazon in 2014, charges clients based on the number of function invocations or the duration of function execution in 100ms intervals [3].

2.2 Security Primer

Microservice security is not a well-studied topic. Currently, the research dedicated explicitly to microservice security is very scarce. However, the core security principles hold for microservices as well as for any other architecture. A variety of modern security protocols and security best practices can be used when building microservice architecture.

This section starts with explaining the basic security concepts of identification, authentication, authorization, access control, and threat modeling. Next, an overview of the security standards related to public key infrastructure and delegated authorization and authentication is provided. This information is necessary for understanding the research papers in Chapters 4 and 7. The section continues with the brief explanation of SOA security standards that are relevant for Chapter 4. The section ends with a short overview of the preventive, detective, and corrective security measures that used throughout Chapters 4, 5, 6, and 7.

2.2.1 The Core Security Concepts

Identification, Authentication, and Authorization

Identification is a process of claiming an identity of a particular entity without a proof, e.g., providing a username. *Authentication* is a process of confirming the claimed identity. To be authenticated, an entity provides a proof of identity to the authenticating party for verification. The proofs of identity, also known as authenti-

cation factors, are based on knowledge, ownership or inherent properties. Examples of authentication factors are 1) passwords, PIN-codes, answers to security questions; 2) ID card, hardware and software security tokens; 3) biometric identifiers such as signatures, handwriting, fingerprints, face, voice, or typing patterns. Authentication that involves more than one factor is called a multi-factor authentication.

Authorization is a process of granting rights to an authenticated entity and specifying what a subject can do. For example, administrative users have more rights available to them than regular users. When a regular user logs in into a system, he/she will not be able to perform administrative tasks such as removing other users from a system. Authorization can also be viewed as the specification of access policies [56].

Access Control

Identification, authentication, and authorization are used to provide access control in computer systems. Multiple ways of imposing access control exist. In an Access Control List (ACL) model, a list of permissions attached to a system object is used to grant access to this object for a specific entity. In Role-Based Access Control (RBAC), in contrast to assigning permissions directly to entities, the entities are assigned particular roles based on the functions they perform. Attribute-Based Access Control (ABAC) extends the RBAC model by allowing additional attributes for higher control granularity. Other access control models exist.

Confidentiality

Confidentiality is a property that information is not disclosed to unauthorized entities during the information lifecycle. Information may need to be kept confidential in transit and storage, as well as destroyed securely. While encryption is used to protect confidentiality of data, other techniques such as message authentication codes (MAC) or digital signatures are needed to ensure data integrity and authenticity.

Threat Model

A *vulnerability* is a weakness in a system that can be exploited by a malicious party. Security threats are usually caused by an exploit of a vulnerability, although other causes such as social engineering and natural disasters are possible. Since it is impossible to be protected from all known and unknown threats, it is necessary to identify possible attack vectors, prioritize the security threats, and plan preemptive actions to avoid the most likely ones. This process is called *threat modeling*.

An *attack vector* is a component of a system that an attacker can tamper with, such as input fields and interfaces, to gain further strategic or financial advantage. An *attack surface* of a system is the sum of all the existing attack vectors. Attack surface reduction is a known security measure.

2.2.2 Public Key Infrastructure

Public Key Cryptography

In symmetric cryptography, the same secret key is used for both encryption and decryption, and the encryption and decryption functions are similar. AES and 3DES

are the examples of commonly used modern symmetric algorithms that are fast and secure. The shortcomings of the symmetric-key schemes include the difficulty of secure key distribution, large number of keys ($n \cdot (n-1)/2$ for a network with n users), and no protection against cheating by the involved parties [95, p.150].

Public key cryptography, or asymmetric cryptography, that addresses the shortcomings of the symmetric cryptography was introduced by W. Diffie, M. Hellman and R. Merkle in 1976 [23]. Following the established convention, let us consider the two parties, Alice and Bob, trying to exchange secure messages using a public-key cryptosystem. To encrypt a message for Bob, Alice should use Bob's public key. To decrypt the message sent by Alice to Bob, Bob should use his private key. Each party maintains a pair of keys, instead of a single key. In addition to the keys used for encryption and decryption being different, the encryption and decryption functions are also different.

The public-key algorithms are based on the notion of a one-way function such that encryption is computationally easy, but decryption is computationally hard. The main one-way functions are based either on the integer factorization problem (RSA) or discrete logarithm problem (DSA, ECDH).

Symmetric encryption is much faster than asymmetric one, but it fails to provide non-repudiation and secure key establishment. Therefore, most practical cryptographic protocols are hybrid protocols that rely on both symmetric and asymmetric algorithms. SSL/TLS protocols, the cornerstones of secure Internet communication, belong to this category.

Transport Layer Security

Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL) are a family of cryptographic protocols designed to provide communication security (prevent eavesdropping, tampering, and message forgery) over a computer network. TLS incorporates many different algorithms with configurable parameters for secure key exchange, encryption, message authentication and integrity.

The plurality of available options and presence of unsafe legacy components often lead to insecure configurations. Multiple open source TLS programming libraries such as OpenSSL and GnuTLS exist. Neither the protocol nor its implementations are perfect. A list of known attacks against TLS/SSL up to 2015 can be found in RFC 7457 [117].

The newest version of TLS, version 1.3 [105], has been under active development since 2014. It removes obsolete and insecure ciphers and hash-functions from TLS 1.2 while, among other things, introducing new security features and improving performance (1-RTT and 0-RTT handshakes).

Digital Certificates and Public Key Infrastructure

While public-key schemes do not require a secure channel, they do require an authenticated channel for the public keys distribution [95, p.344]. An authenticated channel is needed to prevent man-in-the-middle (MITM) attacks where an attacker pretends to be a legitimate communicating party to each of the sides by tampering

with the communication channel.

Digital certificates are a solution to the problem of public keys authenticity. Certificates bind a public key to a specific identity by applying digital signatures. Certificates have a complex structure and include various fields such as a period of validity, issuer, and purpose. X.509 [16] is a cryptographic standard that defines the format of public key certificates that is widely used, also for TLS.

To verify a signature of the given message, a receiver of the message must use the public key of the sender. An entity called Certification Authority (CA) is a mutually trusted third party that issues certificates to the communicating parties. CAs form a chain of trust. Together, CAs and supporting mechanisms create a Public-Key Infrastructure (PKI). Running a real-world PKI is nontrivial. One of the most challenging tasks of PKIs is certificate revocation. The shortcomings of PKI are discussed in various sources [28] and should be acknowledged by the system architects.

We will look at TLS and self-hosted PKI for microservices in Chapter 4.

2.2.3 Delegated Authorization and Authentication

Delegated authorization and shared authentication are an essential part of modern web security. The most tangible and visible mechanism is social login, which is supported by many web services today. However, delegated authorization and shared authentication protocols have many interesting applications for securing inter-service communication in microservice architectures. The following discussion summarizes the security challenges of the popular security protocols underlying delegated authorization and shared authentication. The OAuth and JWT security standards discussed here are important for understanding Chapter 4.

OAuth 1.0

Identity and access delegation, as an act of empowering to act for another, is an important aspect of computer security. OAuth is a delegated authorization protocol providing third-party applications with delegated access to protected resources on behalf of a resource owner.

OAuth 1.0 and OAuth 1.0a [44] include two sets of credentials with each client request: one set to identify the resource owner and another to identify the client (third-party application) itself. Before a client can make authenticated requests on behalf of the resource owner, it has to obtain permission from the resource owner. After the permission is obtained, the authorization server issues a security token called the *access token* to the client. The access token and a related token secret represent the resource owner approval and are used to associate clients' requests with the resource owner.

The client credentials consist of a unique identifier and an associated shared-secret or RSA key pair used for signing. While signing in OAuth 1.0 enables client authentication and message integrity on the application level independently of TLS, confidentiality matters must still be accounted for.

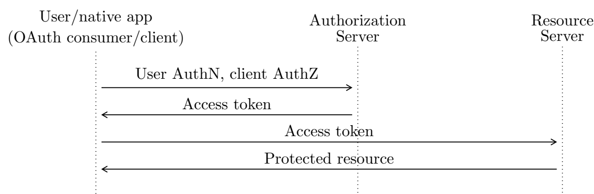


FIGURE 2.2: *OAuth 2.0 Implicit Flow, based on [45, Sect.4.1].*

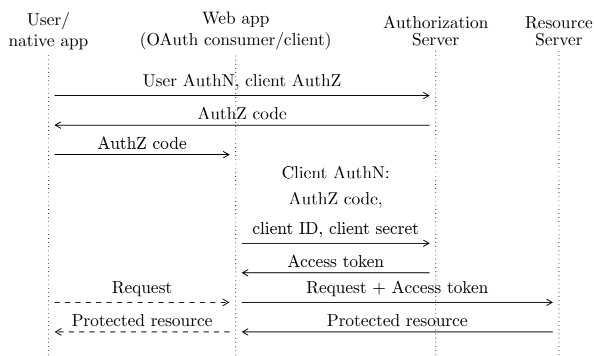


FIGURE 2.3: *OAuth 2.0 Authorization Code Flow, based on [45, Sect.4.2].*

OAuth 2.0

OAuth 2.0 [45] only utilizes access tokens to identify a resource owner; therefore tokens are no longer bound to any particular client. Since OAuth 2.0 does not sign requests, it is notably simpler than OAuth 1.0, but it is completely dependent on TLS. Multiple optional components in OAuth 2.0 have led to interoperability issues between different providers. For example, no specific access token types are required: developers can choose between two incompatible token formats, namely Bearer (or JWT bearer tokens) and MAC tokens.

OAuth 2.0 defines four authorization flows for different use-cases. The two most commonly used flows are Implicit Flow and Authorization Code Flow shown in Figures 2.2 and 2.3. The Implicit Flow is mainly designed for public clients which cannot securely maintain confidential data, whereas the Authorization Code Flow targets trusted clients.

The Authorization Code Flow is more secure than Implicit Flow because the access token is not exposed to a native application, but only to a web application running on a web server. As shown in Figure 2.3, the use of client ID and client secret to authenticate the web application to the authorization server is required before exchanging the authorization code for an access token.

In the context of native clients, the choice between the flows depends on the system architecture. The Implicit Flow suits browser-based applications (JavaScript

applications running in a browser) and native standalone applications that interact with an OAuth provider directly. The Authorization Code Flow targets web applications running on a trusted web server, as well as distributed applications where a backend server proxies requests of native applications and attaches the corresponding access token.

Common Security Pitfalls in OAuth

Developers often fail to implement OAuth correctly due to its ambiguity and complexity. An extensive study of OAuth usage in mobile applications performed by Chen et al. [13] in 2014 shows that 24% of analyzed mobile applications used OAuth and almost 60% of them contained security critical mistakes in design and implementation. Confusing authentication and authorization, and preferring less secure option are among the leading causes of vulnerabilities in OAuth.

Confusing authentication and authorization. OAuth 2.0 is not an authentication protocol for clients but is often incorrectly treated as such. Although user authentication is a necessary step when granting permissions to the client, no continuous client authentication is provided. Wang et al. [131] first identified a dangerous misuse of the access token as proof of client authentication to backend servers in distributed environments. On the other hand, some of the developers who realize the problem try to bridge the authentication gap themselves, which leads to insecure home-brewed OAuth-based authentication protocols, especially in mobile applications [13].

Preferring less secure options. With MAC tokens, each request from the client to the resource server is authenticated based on a symmetric key shared between the parties. Although MAC tokens provide better security than Bearer tokens, MAC tokens are rarely used. As of 2014, only three out of twelve major OAuth providers supported MAC tokens [50].

Developers often ignore the fact that Authorization Code Flow is more secure than Implicit Flow. Using the Implicit Flow for distributed applications with a backend server is quite common. Whether it is a conscious decision to avoid additional work or decision out of ignorance is often unclear.

Common implementation mistakes [13, 122] are the reuse of the authorization code, not asking for the user's consent explicitly, absence of TLS, bundling client secrets with mobile applications, and insecure redirection handling in mobile applications. Interestingly, OAuth security issues arise not only from ambiguities in the standard but also from SDKs promoting insecure choices [131].

OAuth-based Single-sign-on

OAuth 2.0 is used as an underlying layer for shared authentication protocols and Single-Sign-On (SSO) systems. Prominent examples are OpenID Connect, Facebook Login, and Sign In With Twitter. In such schemes, the user authenticates into a third party service (a Relying Party or RP) using a digital identity at an Identity Provider (IdP) of the user's choice. However, additional steps must be taken to

use OAuth 2.0 for authentication. Modern OAuth-based SSO systems mainly utilize token-based authentication, which is inherently vulnerable to token hijacking if the channel is compromised.

OpenID Connect [112] is a widely deployed shared authentication protocol that currently powers log-in systems at PayPal, Google, Yahoo, Stack Exchange and others. OpenID Connect implements authentication on top of the OAuth 2.0 authorization process with minimal changes required. When OpenID Connect is used, each request includes an OAuth 2.0 access token and an OpenID Connect token, called an ID token. The ID token has a standardized format based on JWT and contains information regarding the authenticated user signed by the identity provider.

To support higher security requirements the OpenID Connect specification defines several optional security enhancing properties, among them:

- The RP can request specific information, such as an email, phone number, address, and authentication time, to be returned from the IdP and/or included into the ID token [112, Sect.5.5].
- Signing and encryption of ID tokens sent by IdP, RP authentication, and RP's OAuth authorization requests [112, Sect.6,10].
- More advanced methods of RP authentication to IdP: digital signatures with pre-registered public key or HMAC with a client secret as a shared key. The default method is the HTTP Basic authentication scheme with a client secret [112, Sect.9].

There are no studies of these advanced security options in real world OpenID Connect providers. Serious security flaws were discovered in the design of OpenID Connect extensions for dynamic server discovery and client registration [77].

2.2.4 SOA Security

Over the past two decades, the topic of SOA security has been extensively discussed in literature [41, 87, 119]. SOA security comprises of general security standards and Web Services security standards, including XML security standards.

WS-Security

The WS-Security specification [84], released by OASIS in 2004, offers a common format for security in a SOAP message by utilizing the header part of the XML message to pass along security information. WS-Security incorporates XML Signature [6] and XML Encryption [51] to enforce integrity and confidentiality providing so-called end-to-end security, as well as use of security tokens to establish the sender's identity. WS-Security can optionally be used on top of TLS.

The WS-Security specification describes the framework for securing SOAP messages, leaving the additional functionality to a broad set of specifications, including WS-Trust, WS-Federation, WS-Authorization, WS-Privacy, WS-SecurityPolicy,

WS-SecureConversation, jointly called *WS-* stack*. Each of this WS-Security extensions is a topic of its own.

The WS-* stack has undeniable benefits in terms of modularity and available features, and it provides security in a comprehensive and expandable manner. If implemented correctly, WS-Security successfully eliminates many security threats such as network eavesdropping, message tampering, and message routing [49].

Complexity is an enemy of security. Ironically, the most criticism of WS-* stack relates to its complexity. It is challenging for developers to manage the stack properly, even with the existing tools aimed at hiding its complexity. Multiple implementation vulnerabilities [8] and attacks, such as XML-encryption attacks [61] and XML signature wrapping attacks [73], has been found. Tools like WS-Attacker[4] can assist in Web services penetration testing.

Several studies have confirmed that TLS outperforms WS-Security. The performance of three different security approaches, (1) X509 Token Profile, (2) WS-Secure Conversation, and (3) vanilla TLS, has been compared with (4) the absence of any security mechanisms (message routing only) in [63]. The found ratio in the number of messages processed per second was approximately 1 : 2 : 8 : 14, which clearly shows the advantage of TLS. These results were later confirmed in [29], where the authors showed that TLS remains an order of magnitude more efficient compared to the best WS-Security optimization.

2.2.5 Security Measures

Security of distributed systems, as well as computer security in general, has many dimensions. It matters how the system is built and how it is maintained.

Preventive measures

The long history of security breaches, small and big, shows the importance of addressing the security concerns on early stages of system design and development. Such an approach is called *security by design* and incorporates various basic security principles discussed below.

- *Minimizing attack surface area*, as in reducing the number and scope of components an attacker can interact with, such as system inputs and interfaces, makes a given system harder to exploit.

- *Security by default* embraces the fact that the default system configurations and settings are often used and rarely changed because of convenience, lack of awareness or competence, or other factors. Therefore, the default options should be the secure ones. For example, denying all incoming traffic unless otherwise specified for a given IP-address is safer than allowing all traffic except from explicitly blacklisted IP-addresses. Another example is to require a new user to create a password of at least the specified length.

[4] http://sourceforge.net/projects/ws-attacker/

- *Defense in depth* implies that no component can be trusted and as much as possible should be verified. Similarly to how inputs should be validated in methods, components should not blindly trust each other in a distributed system. Defense in depth is closely related to another principle called *layering of security mechanisms*.

- *Least privilege principle* is concerned with limiting the abilities of an entity to the bare minimum required for performing the relevant tasks. While the principle is often associated with limiting file system permissions, the principle expands further to system resource permissions such as network access and CPU and memory allowance. As we will see in Chapter 4, microservice architecture facilitates the adoption of the defense in depth and least privilege principles.

Detective measures

In order to fix a security issue, it first needs to be detected. Detection of security accidents is a vital system functionality. System and network monitoring, including firewalls, intrusion detection (IDS) and intrusion prevention (IPS) systems, are typical examples of detective measures.

Honeypots are a deception mechanism to detect and deflect unauthorized access to the system by exposing decoy system components and resources in a carefully controlled environment. Honeypots can distract attackers from valuable components and assist in worm detection, worm countermeasures, and spam prevention [102]. Honeypots can be used as research tools and to facilitate understanding of the possible attack landscape [11]. Security testing, including penetration testing, can be considered to be a detective measure.

Corrective measures

The goal of corrective measures is to limit the damage from a security incident. Examples of corrective measures are a software update released to fix a newly discovered security bug or restoring the system from a backup. The forensics process is simplified by having proper logging and system monitoring in place.

The time between a security incident being detected and correcting actions implemented is of critical importance. The longer the response takes, the more damage an attack is likely to create. More information about Intrusion Response Systems (IRS) can be found in a survey by Stakhanova et al.[121].

Need for Security Automation and Self-Protection

While sophistication of attacks is continuously increasing, the required technical knowledge to perform an attack does not follow the same trend. This is mostly because of automation of the attack process and availability of attack tools, also for sale.

Figure 2.4 outlines the relationship between attack sophistication and the required knowledge from 1980 to 2018. We extended the original version [71] that

stops at the year 2000. The figure depicts a small subset of a vast development in the area of computer security from an attacker perspective.

Detective and corrective measures as well as the basics of security automation in context of microservices are the central topics of Chapter 6.

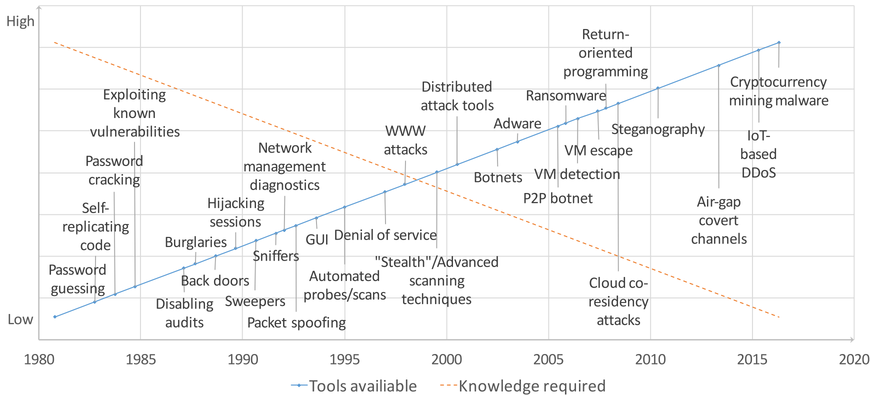


FIGURE 2.4: *The sophistication of attack tools versus required attack knowledge.*

2.3 On Microservice Integration Styles

The integral part of all distributed systems is interprocess communication. In contrast to processes within the same environment that can utilize shared memory, components of a distributed system communicate over a network by sending and receiving messages. A great variety of communication protocols exist.

There are no clear guidelines for how to do service integration, i.e., how services should communicate. As an architectural style, microservices do not limit the possible communication models and integration options. However, the choice of integration solution has an impact on microservice principles and may undermine the benefits of using microservices in the first place.

For example, centralized databases, one of the main application integration styles [48], do not align well with the microservice principles of loose coupling, decentralization and state ownership. Microservice integration through shared databases is likely to negatively impact the system ability to scale and make it more fragile, and should, therefore, be avoided. A recommended database design for microservices [107, p.7] is that each service has a separate database.

The standard choice for microservice integration [85, p.55] is the REpresentational State Transfer (REST) APIs [33]. The goal of this section is to evaluate how the REST style fits with the microservice principles and compare it with other integration solutions. Our main evaluation criteria are the core microservice principles of loose coupling and “smart endpoints and dumb pipes” because the integration

choices affect them the most. We are particularly interested in internal service-to-service communication.

2.3.1 What is REST

REST is an architectural style that defines the behavior of web agents and allows the web to scale. Since its introduction by Fielding & Taylor [33] in 2000, the REST style gained broad adoption. It became a competitor to SOAP and related set of Web Services standards [100].

The REST style is usually defined through the set of *constraints*, specifically client-server, stateless, cache, uniform interface, layered system, and code on demand. The six core design principles of REST as formulated by Erenkrantz et al. [30] are:

- The key abstraction of information is a resource, named by an URL.
- The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes.
- All interactions are context-free.
- Only a few primitive operations are available.
- Idempotent operations and representation metadata are encouraged in support of caching.
- The presence of intermediaries is promoted.

However, a lack of detailed design guidance led to a dissonance between the style and implementations [20, 30, 34]. An absence of dynamic representations of the original message, no definition of a consistent namespace, and misunderstanding of "pipelining" are among known practical problems [30]. Session management and use of cookies are an integral part of the modern web that violates the stateless constraint of the REST style [53]. Consequently, the label "RESTful API" is often put on APIs that are not RESTful [104]. Security has not been addressed by the REST style, which results in much confusion still. The topic of RESTful security is discussed in detail in Chapter 7.

Khare & Taylor [58] identified three limitations of REST in decentralized environments; first, one-shot requests and no recovery mechanisms; second, the only possible communication pattern is one to one, not one to many; third, the absence of bidirectional communication, a client initiates communication. To address the limitations, the authors proposed an extension of the REST style for decentralized systems called the ARRESTED style. However, the style is not widely adopted.

Despite multiple shortcomings of the REST style and future developments in academia, the REST style became dominant for web APIs.

2.3.2 Loose Coupling

Coupling can be defined as "a measure of interdependencies between modules, which should be minimized" [129]. Decomposition of software into modules with high cohesion and low degree of coupling is likely to improve reliability by limiting failure propagation as well as allow systems to scale easier. Loose coupling is one of the core principles of high-quality software design. The importance of low coupling has been recognized in the context of structured development techniques, as well as in object-oriented design [10]. Loose coupling is an essential property of microservices [85, p.30]. Hence, the chosen integration style should contribute to loose coupling.

Coupling Metrics

Although many attempts have been made, there is no *standard* quantifiable metric for coupling of software components. In the context of service communication, tight coupling is often contrasted to loose coupling, but it is not a binary property. Eugster et al. [31] differentiates between the following three dimensions in the context of distributed interaction and publish-subscribe systems:

- *Space decoupling.* The interacting parties are not connected directly and do not hold a reference to each other, but use an intermediary instead.
- *Time decoupling.* The interacting parties do not need to be available at the same time.
- *Synchronization decoupling.* The interacting parties do not get blocked for the interaction duration.

Pautasso & Wilde [97] proposed a set of qualitative coupling metrics for service design. Their paper compares RESTful HTTP, Remote Procedure Calls (RPC) over HTTP, and WS-*/ESB technologies based on twelve facets of loose coupling: discovery, identification, binding, platform, interaction style, interface orientation, model, granularity, state, evolution, generated code, and conversation. The authors conclude that no technology satisfies all loose coupling criteria.

(De)coupling Properties of the Main Integration Styles

Although the REST APIs are the default choice for microservice integration [85, p.55], it is an open question if the REST style is actually required and even sufficient [138]. In practice, a variety of technologies and protocols are used for microservice communication [99].

REST APIs, like any other HTTP-based APIs, can be accessed asynchronously via resource polling or webhooks. WebSockets is another way of achieving full-duplex communication over HTTP. RPC solutions, such as gRPC[5], are also used for microservices. gRPC provides bidirectional streaming while relying on HTTP/2

[5] https://grpc.io

for transport and using Protocol Buffers as a binary serialization toolset and interface description language. On the other side of the complexity spectrum lie messaging systems such as Apache Kafka[6] and RabbitMQ[7] that became widespread in the microservice world.

Table 2.1 compares the popular integration alternatives based on coupling facets from Eugster et al. [31] and selected complexity properties. Asynchronous interaction is one of the intuitive ways of achieving loose coupling. A strong need for asynchronicity arises when the operation requested takes a considerable amount of time. However, asynchronous interaction is more difficult to implement and debug than synchronous communication which results in more complex services. Some of the challenges include a need to correlate requests and responses and deal with timeouts.

The REST style can be used for both blocking and non-blocking synchronous interaction. The latter is possible through long polling. A temporary resource can be created instead of an actual one. A 202 (Accepted) response code can be returned to inform a client that the request is received, but the response is not available yet. The client's responsibility is to check on the temporary resource until a URI to the actual resource is given.

While the publish-subscribe systems offer the most in terms of loose coupling [31], they promote centralization and can potentially become a single point of failure. Decentralization is a fundamental characteristic of microservices. As shown in Table 2.1, asynchronous communication comes at the cost of increased complexity of clients and/or infrastructure.

Interface Coupling

An important coupling metric that does not naturally fit the above-listed classifications is interface coupling. It is a design-specific property that includes:

- *Number of interfaces.* One aspect of service complexity is its surface area that can be defined by the number of external service dependencies. The more dependencies the given service has, the more tightly coupled it is. It is common

[6] https://kafka.apache.org
[7] https://www.rabbitmq.com

TABLE 2.1: *Decoupling facets and related complexity of the main integration styles.*

Interaction paradigm	Decoupling			Complexity	
	Space	Time	Async	Infrastruct.	Service
REST/RPC	No	No	Producer	Low	Low
REST w polling / Async RPC	No	No	Yes	Low	High
Messaging	Yes	Yes	Yes	High	High

to have composed microservices (mashup services) that aggregate information from other services. The disadvantage of multiple interfaces is increased complexity and higher risk of circular dependencies. The "do one thing" principle of microservice design implicitly encourages a low number of dependencies.

- *Frequency of interface use.* If microservices are strongly dependent on each other and do network calls for each operation, such services can be said to be tightly coupled due to a suboptimal service composition. Merging such services into one can be a viable option.

- *Interface evolution.* The "evolutionary design" principle of microservices implies a frequent modification of interfaces, both semantical and syntactical. In such a dynamic environment, breaking changes and backward compatibility becomes a serious concern. GraphQL[8], a query language for APIs, can be leveraged in API Gateway implementations to solve the problems of versioning and over- and under-fetching.

2.3.3 How Dumb Are the Pipes?

The previous subsection has demonstrated that loose coupling of services in time, space and synchronization domains may lead to tighter coupling with specific infrastructure components such as messaging systems and make the services more complex. It is now necessary to explain how the complexity is distributed between the services and the mechanisms connecting them.

"Smart endpoints and dumb pipes" is one of the microservice characteristics. According to Lewis & Fowler [38], all the communication logic should be contained inside the services and the communication itself should be done via lightweight mechanisms. Moving the complexity into the services can also be seen in the popular Circuit Breaker pattern that prevents cascading failures by adjusting the service behavior.

Complexity Scale

The REST style assumes no logic in the network. Synchronous request-response communication pattern makes REST APIs the most unadorned "pipe". Specifically, HTTP requests with parameters, no or a minimal state, certainly no memory across transactions (apart from a database) and no delivery guarantees contribute to the overall simplicity of the REST style.

More complicated options, such as various message-oriented middleware and Enterprise Service Bus (ESB), are located on the opposite side of the spectrum. ESBes can provide support for naming, location, service discovery, replication, protocol handling, communication faults, synchronization, concurrency, transactions, storage, access control, and authentication. However, messaging systems do not need to provide all above-listed features. The existing service integration options

[8] http://graphql.org

FIGURE 2.5: *A relative complexity scale of the existing service integration solutions.*

can be placed on a complexity scale [24] based on the number of features and functionalities provided. An example of such a scale is given in Figure 2.5.

Complexity in Messaging Systems

To get a more robust understanding of the complexity scale, we look at the differences between the two most widely adopted open-source messaging systems, namely Apache Kafka and RabbitMQ. Dobbelaere & Esmaili [24] performed an extensive comparison of the functionality of these two systems as of 2017. Their work creates a basis for Table 2.2 that summarizes the core properties. As shown in Table 2.2, RabbitMQ offers a much more comprehensive set of functionalities when compared to Apache Kafka, and, therefore, is more complex. In the microservice world, highly intelligent "pipes" are unwanted: microservices favor simplicity over complexity.

2.3.4 Security, Performance and Engineering Cost

Despite the conceptual fitting of the given integration style with microservice principles, there are several practical considerations to address when choosing a specific integration solution.

SECURITY. Different solutions have different embedded security features. The REST style does not provide any guidance on the security mechanisms and leaves the security matters entirely at the discretion of developers. In contrast to REST, solutions such as gRPC, Kafka, and RabbitMQ support some security features out of the box, including mutual TLS for entities authentication and traffic encryption. As for early 2018, Kafka and RabbitMQ support various pluggable SASL mechanisms and authorization options.

PERFORMANCE. The distributed nature of microservices is the cause of its low performance when compared to non-distributed systems. Interoperability and decentralization come at the cost of reduced performance since it is suboptimal compared to homogeneous and monolithic environment. In many cases, high-performance computing implies tight coupling: network calls are slower than system calls, communication protocols with binary formats (Protobuf) are faster compared to XML- or JSON-based ones. The latter implies, for example, that for time-sensitive tasks gRPC is a better fit than the REST APIs.

ENGINEERING COST. Availability of tools to simplify the development process, rapid prototyping, testing, and maintenance are important factors. Amount and

TABLE 2.2: *Comparison of the features of Apache Kafka and RabbitMQ*

	Complex routing	Delivery, at least once	Replication	Transactions	Multicast	Dynamic scaling	Long term msg store	Msg replay	Multi-protocol	Distrib. topology	Multi-tenancy, isolation	Consumer tracking	Disk-less use	Publisher flow control	Queue size limit	Msg TTL
Apache Kafka	Y	unord.	Y	N	customer side	Y	Y	Y	N	N	N	N	N	N	N	N
Rab-bitMQ	N	ord.	Y	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y

quality of the documentation and conciseness of the code are essential. Existing infrastructure compatibility and vendor lock-in are relevant concerns.

2.3.5 Discussion

ARE DECENTRALIZED PROTOCOLS A BETTER FIT? As shown earlier, publish-subscribe solutions promote loose coupling but lead to more centralization by introducing an additional infrastructure component. One potential way to facilitate decentralization is to use gossiping protocols for network self-management. However, the small message sizes and relatively slow message propagation [9] make such solutions inappropriate for microservices where message exchange rates are high and non-uniform.

THE RELATIONSHIP BETWEEN REST AND MICROSERVICES. REST APIs, as well as any HTTP-based APIs, are easy to implement due to the prevalence of the HTTP protocol and supporting libraries that are readily available for almost all programming languages and platforms. REST APIs are easy to debug and comprehend for humans due to the synchronous nature of HTTP. However, REST APIs are only one of many valid ways to integrate microservices. Moreover, it is not necessarily the one providing highest decoupling: messaging systems are scoring higher on all the decoupling metrics, as shown in Table 2.2.

However, there are cases when synchronous communication in general, and REST APIs in particular, are a better option. Such cases include user authentication and other critical information requests where there is a need to wait for a reply. For interoperability reasons, it makes sense to have REST APIs on the edge services designated for consumption by users and third parties.

COMBINING APPROACHES. What microservice integration style should be adopted is dictated by multiple factors and is often system specific. Preferring REST APIs over ESB will not bring all the microservice benefits by itself; it is still possible to

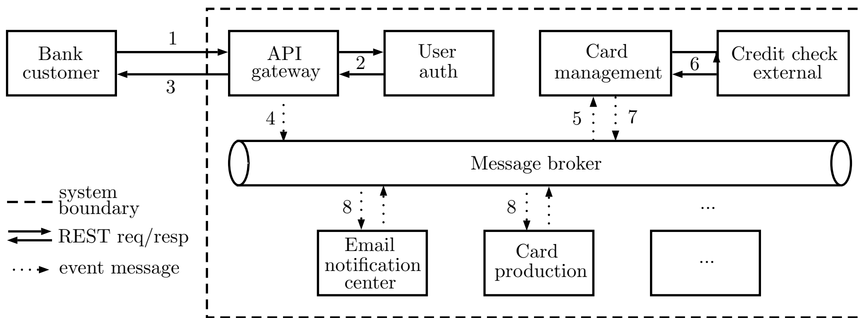


FIGURE 2.6: *A microservice-based bank built using multiple inter-service communication styles.*

unintentionally build a tightly coupled system using REST. Having the right microservice composition and deployment practices may be more important than integration style in some cases.

It is likely that a reasonably sized system will employ several different integration styles. Figure 2.6 depicts the use of synchronous REST API calls alongside event messaging through a message broker when processing a user request for a new credit card. A customer orders a new credit card (1). Upon successful user authentication (2), the customer receives a preliminary order confirmation (3). A new card request event is put into a queue (4). A card management service picks up the relevant event (5) and starts processing it. If an external credit check is successful (6), the card service fires an event requesting an email to be sent to the user and an event initiating a card manufacturing process (7). The corresponding events are received by the email notification center and card production line (8). The system structure is greatly simplified.

2.3.6 Conclusion

Microservice architectural style does not enforce strict rules on the integration patterns and inter-service communication options. However, the core principles that guard the scalability and flexibility properties of microservices, mainly the notion of loosely coupled services and "smart endpoints and dumb pipes" principle, should be accounted for when choosing an integration style.

While the REST architectural style is still a prevalent choice for microservice communication, it is by far not the only one. Multiple technologies exist and favoring one over the other depends on many factors including alignment with the microservice principles as well as practical considerations such as security and performance. The choice of microservice integration style is system specific and should be approached with deliberation since many factors are involved. A combination of different integration technologies is a viable option.

Loose coupling brings high agility to the system. Asynchronous communication is the starting point of decoupling. Although messaging solutions allow build-

ing highly decoupled systems, they may make the infrastructure more centralized. At the same time, the range of publish-subscribe solutions is wide and exhibit different levels of complexity. This evaluation suggests that simpler message queues are a better fit for microservices.

Summary of Papers

3

This thesis is based on four research papers published in various peer-reviewed conference proceedings. A synopsis of each paper is given below.

Paper I: *Overcoming Security Challenges in Microservice Architectures*

Microservice security is a multifaceted problem. This paper provides a taxonomy of microservice security by grouping the applicable security threats and mitigation into six layers: orchestration, service/application, communication, cloud, virtualization, and hardware. We also survey the available literature.

Since security is a trade-off between minimizing the budget and covering more attack vectors, addressing all possible threats is infeasible in practice. We believe it will be most fruitful for the real-world developers to concentrate on the application and communication layers because the other concerns are likely to be outside their control or would require specialized technical training.

The paper identifies five microservice design principles that affect security: do one thing and do it well; automated, immutable deployment; isolation through loose coupling; diversity through system heterogeneity; and fail fast. We argue strongly for the defense in depth approach to microservice design that is facilitated by microservice architecture.

We also survey several prominent microservice security trends in the industry, namely mutual authentication of services using MTLS and principal propagation via security tokens. Both mechanisms belong to the communication layer in our taxonomy. Since no clear trends for microservice authorization exist in the industry, we sketch two possible solutions for fine-grained authorization.

Furthermore, we developed an open source prototype framework (MiSSFire) for establishing trust and securing microservice communication with MTLS, self-hosted PKI, and security tokens. The performance of the framework is evaluated against a toy microservice-based bank system of our design (MicroBank). The performed experiments show that the average performance overhead of all the security mechanisms combined accounts for around 11%. Both MiSSFire and MicroBank are open source projects that are available on GitHub.

Paper II: *Low-Level Exploitation Mitigation by Diverse Microservices*

In this paper, we argue that microservice architecture has inherent security benefits in terms of isolation and diversity. These benefits are a consequence of the distributed nature of microservices combined with the ubiquitous use of virtualization and the practice of frequent redeployment. The paper explains how a low-level attack progression in a virtualized homogeneous environment differs from an attack in a virtualized heterogeneous environment, and how service isolation and software diversity improve security given other conditions unchanged.

We demonstrate the benefits of using a microservice architecture to defend against remote low-level exploitation. To emphasize the added security benefit of the increased control flow isolation, we developed a simple implementation of a microservice network and its monolithic counterpart and performed exploitation attempts against both of them. Our trivial experiment shows that a microservice solution is less vulnerable to low-level attacks than a deployment monolith.

The paper elaborates on how the inherent security benefits of microservices can be improved even further. Maximizing interface security, avoiding unnecessary node relationships, and introducing asymmetric node strength are briefly discussed as the most straightforward security measures. The paper proposes a security monitor, a hybrid of IDS and IPS that is based on N-version programming, for protecting security-critical microservice applications.

Paper III: *A Game of Microservices: Automated Intrusion Response*

There is a growing need for self-protection and adaptive security mechanisms in software systems. Since many attacks are now automated/scripted, manual responses are insufficient, mainly when there are critical resources that can be permanently lost or stolen.

In this paper, we propose an architecture for a cost-sensitive adaptable intrusion response system for microservices called μGE. The system collects information about a microservice network and then plans an appropriate response. μGE utilizes a game theoretic approach to respond to network attacks in real-time automatically.

This paper models the strategic interaction between an attacker and a defender as a finite dynamic two-player zero-sum game. Game theory is a useful foundation for planning responses, since each response may in itself constitute a small loss even though it mitigates a later, larger loss. The number of lookahead steps (depth of the game) can be adjusted based on the available computational resources and security requirements.

We discuss the appropriate defense responses specific to microservices. The defender's actions include but are not limited to service rollback/restart, diversification through recompilation or binary rewriting, diversification through a cloud provider, split or merge services, and isolation/shutdown. This paper builds on

the idea of a security monitor from Paper II.

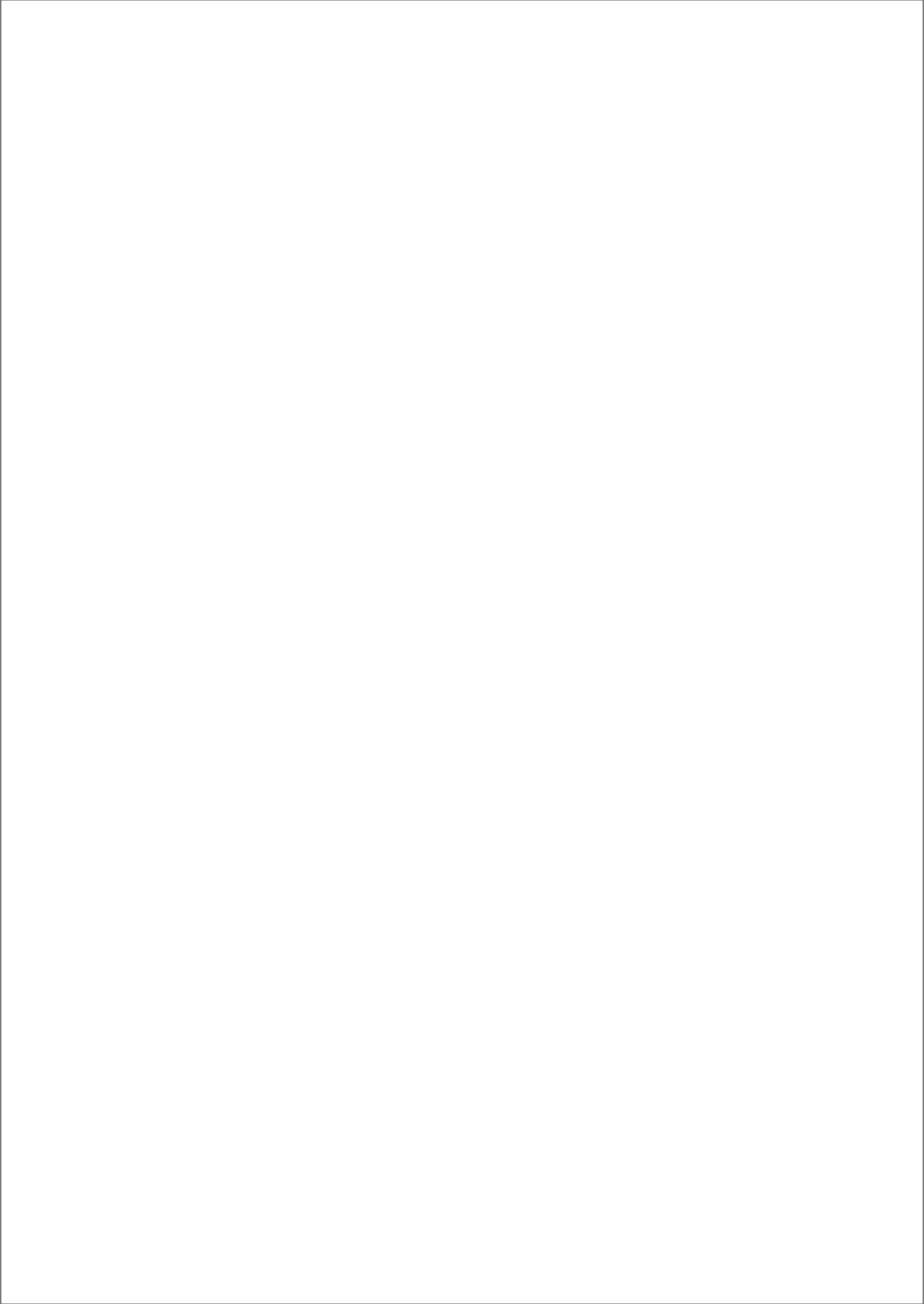
We have a prototype implementation available with a simulated network (as yet unpublished). Preliminary results indicate the following: the system can plan and execute a response, stopping or slowing down the attacker; planning is computationally very intensive for highly connected graphs; it is likely that some or much of the planning can be done in advance; a promising possibility is to use the system to plan or evaluate the defensibility of a microservice network.

Paper IV: *RESTful Is Not Secure*

REST is the default choice for microservice integration. This paper analyzes the REST paradigm from a security perspective and identifies the significant incompatibilities between the style constraints and typical web security mechanisms such as token-based authentication, client-side request signing, and delegated authorization and shared authentication.

REST has not been developed with security in mind. In fact, several core aspects of REST are in direct conflict with security: stateless resources, cacheability, and code-on-demand. In practice, this is often mitigated by not following the style to the letter. We discuss alternative microservice integration approaches in Section 2.3.

Scientific results



Paper I

Overcoming Security Challenges in Microservice Architectures

Tetiana Yarygina, Anya Helene Bagge

4

© 2018 IEEE.

Reprinted, with permission, from: T. Yarygina and A. H. Bagge. "Overcoming Security Challenges in Microservice Architectures". In: *12th IEEE International Symposium on Service-Oriented System Engineering (SOSE'18)*. Bamberg, Germany: IEEE, Mar. 2018, pp. 11–20. DOI: 10.1109/SOSE.2018.00011

Overcoming Security Challenges in Microservice Architectures

Tetiana Yarygina, Anya Helene Bagge

Department of Informatics, University of Bergen, Norway

Abstract The microservice architectural style is an emerging trend in software engineering that allows building highly scalable and flexible systems. However, current state of the art provides only limited insight into the particular security concerns of microservice system.

With this paper, we seek to unravel some of the mysteries surrounding microservice security by: providing a taxonomy of microservices security; assessing the security implications of the microservice architecture; and surveying related contemporary solutions, among others Docker Swarm and Netflix security decisions. We offer two important insights. On one hand, microservice security is a multi-faceted problem that requires a layered security solution that is not available out of the box at the moment. On the other hand, if these security challenges are solved, microservice architectures can improve security; their inherent properties of loose coupling, isolation, diversity, and fail fast all contribute to the increased robustness of a system.

To address the lack of security guidelines this paper describes the design and implementation of a simple security framework for microservices that can be leveraged by practitioners. Proof-of-concept evaluation results show that the performance overhead of the security mechanisms is around 11%.

Keywords *Microservices, defense-in-depth, SOA, distributed systems, cloud, PKI, authentication, MTLS, REST, JWT*

4.1 Introduction

Microservices is an architectural style inspired by Service-Oriented Architecture in combination with the old Unix principle of "do one thing and do it well". Microservices are intended to be lightweight, flexible and easy to get started with, fitting

in with modern software engineering trends such as Agile development, Domain Driven Design (DDD), cloud, containerization, and virtualization. The most frequently listed benefits of microservice architecture [85, 98] are organizational alignment, faster and more frequent releases of software, independent scaling of components, and overall faster technology adoption. The disadvantages include the need for making multiple design choices, the difficulty of testing and monitoring, and operational overhead when compared to typical non-distributed solutions.

The modern use of the term *microservice* dates back to 2011 [38], but the community has so far not reached a full consensus on a formal definition of the style. The fundamental basis of microservices brings together design principles from distributed systems and services, and from classic programming principles of abstraction, modularity, separation of concerns and component-oriented design. Although the underlying principles are widely explored in the academic literature, research on microservices themselves is lagging behind the rapid adoption and development in the software industry.

Understanding the distinctiveness of microservices is crucial. In particular, microservices bring new security challenges, and opportunities, that were not present in traditional monolithic applications. These challenges include establishing trust between individual microservices and distributed secret management; concerns that are of much less interest in traditional web services, or in highly modular software that only runs locally. Effective microservice security solutions need to be scalable, lightweight and easy to automate, in order to fit in with the overall approach and be adopted by industry users. For instance, manual security provisioning of hundreds or thousands of service instances is infeasible. As services are migrated from offline applications and monolithic web services to a microservice-style architecture, code that was never designed to be accessible from outside is now exposed through Web APIs, raising multiple major security concerns.

The past years have seen rapid adoption of microservices in the industry, and yet there has been surprisingly little focus on security. There is a growing body of literature [1, 26] that recognizes the need for microservice security evaluation. Security is one of the greatest challenges when building real-world systems, hence there is an urgent need to address the security concerns in microservice architecture.

Researchers have not treated microservice security in much detail; Fetzer [32] discusses how microservices can be used to build critical systems if the execution is warranted by secure containers and compiler extensions. Otterstad & Yarygina [93] suggests a combination of isolatable microservices and software diversity as a mitigation technique against low-level exploitation. Sun [123] explored the use of an Intrusion Detection System (IDS) for fine-grained virtual network monitoring of a cloud infrastructure based on Software Defined Network (SDN). While being presented as a solution for microservices, in reality it does not exploit any microservice-specific features.

Although the studies by Fetzer [32] and Otterstad & Yarygina [93] highlight the isolation benefits of microservice design, a systematic understanding of how the ar-

chitecture affects security is still lacking. Moreover, microservice security is an overloaded term that requires proper clarification, and there is no overview of emerging industry security practices either.

This paper identifies and unravels some of the mysteries surrounding microservice security, and is the first study to undertake a holistic approach to microservice security. In summary, we make five contributions to the understanding of microservice security:

- putting microservices and their security in the bigger context of SOA and distributed systems (Section 4.2);
- decomposing the notion of microservice security into smaller and more familiar components, with a formalized attack model (Section 4.3);
- analyzing the security implications of microservice design (Section 4.4);
- identifying several prominent microservice security trends in industry (Section 4.5);
- presenting an open source prototype security framework for microservices (Section 4.6).

4.2 What Microservices Really Are

Before we proceed with the security challenges for microservices, it is important to understand how microservices relate to other software architectural styles as well as to agree on the definitions.

4.2.1 Defining Microservices

Numerous terms are used to describe microservices, the most common microservice definitions are presented below:

- Newman [85] defines microservices as small autonomous services build around the following principles: *model [services] around business concepts*, *adopt a culture of automation*, *hide internal implementation details*, *decentralize all things*, *isolate failure*, and make services *independently deployable* and *highly observable*.
- Lewis and Fowler [38] view microservices as *"an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."*

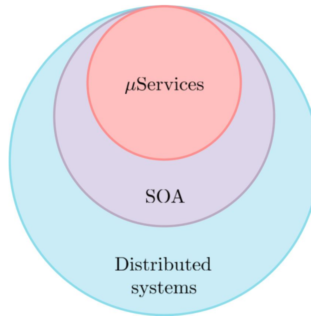


FIGURE 4.1: *Microservice architecture in perspective. Microservices are an implementation approach to SOA. SOA is a subclass of distributed systems.*

The term has also been used in unrelated ways; for example, Kim [59] uses the term "microservice" when referring to the basic security primitives in context of formal methods: authentication microservice, integrity microservice, among others. This definition should not be confused with the modern definition of microservice architecture.

4.2.2 Defining Service-Oriented Architecture (SOA)

SOA can be seen as a predecessor to microservices; Josuttis [57] gives the following definition: "SOA is an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners".

The core technical concepts of SOA are:

- *Services.* Service in the context of SOA is *"an IT realization of some self-contained business functionality"* [57]. Based on multiple existing definitions of a service, additional situation-dependent attributes that services may have are: self-contained, coarse-grained, visible/discoverable, stateless, idempotent, reusable, composable, vendor-diverse.

- *Interoperability.* SOA interoperability in most cases is achieved via the Enterprise Service Bus that enables service consumers to call the service providers.

- *Loose coupling.* Loose coupling is needed to fulfill the goals of flexibility, scalability, and fault tolerance. Loose coupling is a principle that aims to minimize dependencies so the change in one service does not require a change of another service.

An important aspect of SOA, as stated in the SOA Manifesto, is that *"SOA can be realized through a variety of technologies and standards"*.

4.2.3 Déjà Vu

Majority of opinions on microservices fall into one of the two following categories: (1) microservices are a separate architectural style [38, 106]; (2) microservices are SOA [85, 138]. Some sources [26, 99] take a middle ground by viewing microservices as a refined SOA. The frequently mentioned differences are the degree of centralization, size of services, decomposition based on bounded context, and level of independence. However, these criteria are too vague to be used in a rigorous scientific comparison. They are neither easily quantifiable—it is unclear how the service size should be measured, nor sufficient to serve as style constraints.

Based on the above-listed definitions of SOA and microservices it is clear that these two approaches are similar. The definitions of SOA services and microservices are almost identical. This conclusion is important because it means that the security issues faced by practitioners who were migrating to SOA are now relevant for practitioners migrating to microservices. Figure 4.1 depicts this relation.

Zimmermann [138] is the first to systematically compare microservices and SOA, by surveying the authoritative technical blog posts and online articles on microservice principles. After contrasting the distilled microservice principles and SOA characteristics, he concludes that "the differences between microservices and previous attempts to service-oriented computing do not concern the architectural style as such (i.e., its design intent/constraints and its platform-independent principles and patterns), but its concrete realization (e.g., development/deployment paradigms and technologies)" [138].

Hence, we may expect many of the concerns and solutions that apply to SOA to also be applicable to microservices.

4.2.4 Distributed Systems

When talking about microservices many sources repeat the same fallacy of contrasting microservices to monolithic applications. The real situation is not as binary. In practice, a "monolithic" application may be highly modular internally, being built from a large number of components and libraries that may have been supplied by different vendors, and some components (such as a database) may be also distributed across the network. The issues of decomposition, concerns separation, and designing and specifying APIs will be similar regardless of whether API calls are made locally or across the network.

The essence of microservices is that they are (or compose to form) highly modular, distributed systems, reusable through a network-exposed API. This implies that microservices inherit advantages and disadvantages of both distributed systems and web services.

While distributed systems bring multiple highly desirable benefits such as scalability on demand and embrace of heterogeneity, these systems also come with drawbacks. Distributed systems are more challenging to develop in the first place. Moreover, monitoring, testing, and debugging such systems is more difficult than for non-distributed systems. Other well-known problems include maintaining data

consistency and replication among nodes, node naming and discovery due to constantly changing network topology, and dealing with unreliable networks.

Important distinguishing characteristics of distributed systems over monolithic systems can be summarized as follows (adapted from Tanenbaum [125]): 1) The overall system state is unknown to individual nodes; 2) Individual nodes make decisions based on the locally available information; 3) Failure of one node should not affect other nodes. The microservice style enforces these requirements.

4.2.5 Microservices in Context of Other Technologies

A technology closely related to microservice philosophy is unikernels [68]. Unikernels are fixed-purpose OS kernels which run directly on a hypervisor. A full system would consist of multiple unikernels performing different tasks in a distributed fashion. The microservice design principles of small independent loosely coupled single-purpose components fit well into the unikernels world. Unikernels promise to provide high security and strong isolation of virtual machines while still being lightweight like containers.

Some programming languages, such as Erlang and its successor Elixir, support the distributed computing model by design. Erlang is an actor-based programming language. In the Erlang world "everything is a process" and the only process interaction method is through message passing. Erlang language features encapsulation/process isolation, concurrency, fault detection, location transparency, and dynamic code upgrade [4, p.29]. Erlang has advanced monitoring tools, and it is used for several high-load systems such as the messaging service WhatsApp. Systems built with Erlang are inherently microservice-like. As the aforementioned examples show, the core microservice principles can be rediscovered in many technologies.

4.2.6 Summary: Essence of Microservices

Compared to other software and service architectures, these are the distinctive characteristics of microservices:

- *Distributed composition:* Microservices build on other microservices and communicate across the network. (Compared to monolithic services.)

- *Modularity:* Microservices will tend to offer finer-grained APIs that lend themselves to flexible reuse. (Compared to big, single-purpose APIs/applications.)

- *Encapsulation:* Services are to a large degree encapsulated and isolated from others, and may even be written in different programming languages. (Compared to libraries and object-oriented encapsulation.)

- *Network service:* Services are network-accessible, and reuse happens by contacting the service rather than by installing and linking to a library. (Compared to a classic modular design.)

4.3 Layered Security for Microservices

Building secure systems is hard. The classic security objectives are data confidentiality and integrity, entity and message authentication, authorization, and system availability. Naturally, these abstract objectives can be realized in many different ways in practice. How these objectives are met and to what degree are the system-specific architectural decisions that depend on the particular threat model, budget, and expertise. An important issue is where in the system to place security mechanisms.

We argue that microservice security is a multifaceted problem and it relies heavily on underlying technologies and the environment. To get to the bottom of it, we need to decompose microservice security into its components.

4.3.1 Taxonomy of Microservice Security

To illustrate the underlying complexity of the problem, we divided microservice security concerns into six categories or layers as shown in Table 4.1: hardware, virtualization, cloud, communication, service, and orchestration. We do not claim the proposed layering is complete, but we do claim that it provides a good overview of the topic.

This decomposition is based on the basic computer science principles and common sense. While it is inspired by the OSI model and its seven layers of communication, the proposed hierarchical decomposition also incorporates new trends in software engineering such as virtualization and orchestration. This decomposition is applicable to any modern distributed system and is not specific to microservices or SOA. However, it is crucial for the discussion of microservice security.

The decomposition shows that multiple security choices should be made on each level. There are many places where security components reside, and, more importantly, where security can fail. The system should not be treated as a black box. Some levels bleed into each other, e.g. virtualization is a main enabling technology for cloud computing. The separation of security concerns is not always strict.

The two bottom-most layers, hardware and virtualization, are at least partially accessible to an attacker with shell access on the host or virtual machines/containers correspondingly. A malicious hardware manufacturer that provides hardware with backdoors is a threat. On the Cloud level, a cloud vendor itself is a threat. Other tenants are also a potential threat in the cloud using various side-channel attacks, such as the FLUSH+RELOAD technique [133], or Meltdown and Spectre [60]. For communication and orchestration levels, a network attacker inside the perimeter who can eavesdrop and manipulate traffic is a major concern. For the service/application level, the threat of an external attacker should be considered.

The price of addressing the threats on different levels varies. For example, hardware concerns are extremely difficult to address, if at all practically possible. Most developers would be concerned with service/application and communication layers because addressing the security concerns on these levels is cost-feasible for them.

TABLE 4.1: *Proposed Hierarchical Decomposition of Microservice Security Issues into Layers*

Layers	*Threat examples*	*Mitigation examples*
Hardware	Hidden under abstraction, but still a reliability and security concern; hardware bugs are extremely dangerous because they undermine security mechanisms of other layers [103]; hardware backdoors can be introduced at manufacturing time [70].	Designing own hardware; diversification of hardware [70]; use of Hardware Security Modules (HMS).
Virtualization	Deployment affects security; OS processes offer little separation from other services in the same system; containers and VMs offer more protection against compromised services. Attacks include: Sandbox escape, hypervisor compromise, and shared memory attacks; also, use of malicious and/or vulnerable images is another serious security concern [15].	Preferring deployment options with stronger isolation; secure configurations such as no shared library access and no shared hardware cache; verification of image origin and integrity; timely software updates; principle of least privilege.
Cloud	Cloud computing brings a myriad of security concerns [124], including unlimited control of cloud provider over everything it runs; there are few technical options to prevent disruption or attacks from a malicious provider.	Reverse sandboxes protects enclaves of code and data from any remote attack including attacks from OS and hypervisor. SGX (Intel), SME/SEV.
Communication	Classic attacks on the network stack and protocols; attacks against protocols specific to the service integration style (SOAP, RESTful Web Services [100, 134]). Attacks include: eavesdropping (sniffing), identity spoofing, session hijacking, Denial of Service (DoS), and Man-in-the-Middle (MITM); also attacks on TLS: Heartbleed [126] and POODLE [78].	Use of standard and verified security protocols such as TLS or JSON security standards. Security aspects of the chosen service integration style should be considered. Trivial mitigation like not re-using credentials across services.
Service/ Application	Typical and still very common application-level security problems are SQL injection flaws, broken authentication and access control, sensitive data exposure, Cross-Site Scripting (XSS), insecure deserialization, general security misconfiguration. The ten most critical web application security risks are published annually by OWASP [94].	Static/dynamic code analysis, manual code review. Basic software engineering practice: input validation, error handling, clear and well-documented APIs. Protection of the data at rest (encryption). Avoid languages especially vulnerable to, e.g., buffer and integer overflows.
Orchestration	Management, coordination, and automation of service related tasks, including scheduling and clustering of services. Microservice network structure may change continuously due to services being stopped, started, and moved around; service discovery [80, 107] provides a DNS-like central point for locating services. Attacks include: compromising discovery service and registering malicious nodes within the system, redirecting communication to them.	Protection of orchestration platform and its components is critical, but not well-investigated area. A secure implementation of service discovery and registry components is important.

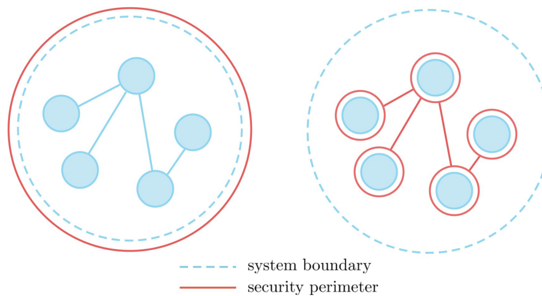


Figure 4.2: *Microservices redefine the notion of perimeter security and move towards defense in depth. The figure excludes infrastructure services, such as API Gateway or monitoring service*

4.3.2 Attack Model: Redefining Perimeter Security

Until recently perimeter defense was the most common approach to security of microservice-based systems [85][p.173]. From the modern security perspective, perimeter security is in general considered insufficient—we should rather assume that the other services in the system may be compromised and hostile ("trust no one"). The rise of microservices, as well as advances in security automation, facilitate placement of additional security mechanisms inside the perimeter (see Figure 4.2). In other words, defense in depth as a concept of placing multiple and sometimes overlapping security mechanisms on different levels throughout the system becomes more feasible with microservices.

We assume an adversary is able to compromise at least one service inside the perimeter and wants to move laterally through the system to gain full control. If internal services blindly trust whoever is calling them, then a single compromised microservice will allow an attacker to manipulate all the other nodes in the microservice network, for example by issuing arbitrary malicious requests that the nodes will fulfill. The latter is sometimes referred to as a confused deputy problem [85]. The adversary can attempt to eavesdrop on the inter-service communication, insert and modify data in transit. We also make a standard cryptographic assumption that the adversary is computationally bound.

Security is a trade-off between minimizing the budget and covering more attack vectors. For practical reasons, we assume the hardware and cloud providers are trusted, as well as the way microservices are deployed provides high degree of isolation. We believe that the given model is the most realistic approximation of the real world and reaches the limits of what software developers and security engineers will be willing to accept in practice. The following discussion is centered around this model.

4.4 Security Implications of Microservice Design

Several important security properties emerge as side effects of microservice design. Herein, we describe and evaluate the properties distilled from conducting a careful

literature review on the subject [26, 38, 85, 98, 99, 106, 107, 138], as well as discussions with practitioners and personal experience. We also provide an interpretation of the implications.

Do one thing and do it well The properties of context boundary, design around business concepts, and Domain Driven Design (DDD) usually results in a smaller codebase per microservice. If we attempt to measure microservices size in lines of code (LOC), then the more LOC there are, the more bugs the service has. LOC and bugs are statistically correlated. More bugs in general means more exploitable bugs in particular. Less complex code is easier to maintain. A smaller codebase of individual microservices results in a smaller attack surface given the other conditions, such as the overall code quality, remain the same. This statement can be further supported by reduced cognitive complexity of the code that facilitates better code comprehension for individual developers. This is not necessarily true for system architects who still need to maintain a more global view of the system.

Automated, immutable deployment Services should be immutable: to introduce permanent changes to microservices, services should be rebuilt and redeployed. Microservices immutability improves overall system security since malicious changes introduced by an attacker to a specific microservice instance are unlikely to persist past redeployment. Automation should be leveraged in maintaining the security infrastructure. Immutability aids the security of microservices similarly to how immutability promotes correctness in programming languages [14].

Isolation through Loose Coupling Both SOA and microservices are built around the concept of loose coupling. Microservices take the concept even further by emphasizing the share nothing principle and strict data owning. This implies that each service can be isolated: only able to access the information it needs, and only able to access the particular services it needs. This limits the damage should an individual service be compromised.

Better isolation as an inherent security benefit of microservice design has been discussed by Otterstad & Yarygina [93]. Fetzer argues [32] that microservices can be used to build critical systems where integrity, confidentiality, and correct execution inside microservices is warranted by secure containers and compiler extensions. Such secure containers were implemented as Docker containers using an Intel Software Guard Extensions (SGX) enclave that can protect the microservices running inside such secure containers from OS, hypervisor, and cloud provider level attacks—with a degree of isolation for each component that is much greater than what is achieved in typical monolithic applications.

Diversity through System Heterogeneity Distributed systems are often heterogeneous systems. Microservice architecture embraces this fact by allowing the individual components to be written in any programming language and/or technology given that they retain same interfaces (service contracts). This results in natural diversity of components in the microservice architecture.

Otterstad & Yarygina [93] suggested diversification of microservices as a miti-

gation technique against low-level exploitation. Although system diversity as a security inducing property is not new, the application of it in microservice setting is. Diversity in computer systems can be achieved in many ways including the use of different programming languages, compilers, OS/basic images. Microservice design philosophy readily allows for such approaches to be taken. Even coexistence of older and newer versions of the same microservice adds to the heterogeneity of the system. Otterstad & Yarygina [93] has also suggested use of N-version programming to improve microservice security.

Fail Fast Although fault tolerance does not directly translate to security, we believe it contributes enough to be listed. If the main point is to disrupt the service, such as in case of Denial of Service (DoS) attacks, fault tolerance does directly translate to security. In contrast to monolithic systems where a failure is often total, distributed systems can be characterized with partial failures where only some of the nodes fail. A microservice network should tolerate the presence of partial failures and limit their propagation. The Circuit breaker pattern [80] prevents cascading failures and increases overall system resilience by adjusting the node behavior if the network interactions with its peers fail partially or completely. Following the fail fast principle will decrease the likelihood of attacks succeeding and minimize the possible damage. Fundamentals of fault tolerance in distributed systems can be found in a book by Tanenbaum [125].

4.5 Emerging Security Practices

Although there are currently few industry practices for microservice security, some interesting trends present themselves. The first one is the use of Mutual Transport Layer Security (MTLS) with a self-hosted Public Key Infrastructure (PKI) as a method to protect all internal service-to-service communication. The second trend is use of tokens and local authentication. Both approaches are leaning towards the defense in depth approach to security and further support our attack model presented in Section 4.3.2.

4.5.1 Mutual Authentication of Services Using MTLS

While sharing the same underlying concept, two different solutions for establishing trust between microservices were developed simultaneously by Docker and Netflix.

Docker Swarm Case

Docker Swarm is a container orchestration solution and clustering system for Docker that allows building distributed systems. Docker Swarm is particularly interesting because a) it is a popular platform for implementing microservices; b) it has a variety of built-in security features. MTLS is used by all the nodes in a swarm to authenticate each other, encrypt all the network traffic, and differentiate between worker and manager nodes [79, 90]. Docker Swarm automatically deploys its own PKI to provide identity to all the nodes.

The first manager node generates a new self-signed root Certificate Authority

(CA) along with a key pair. A hash of the root CA and a randomly generated secret are sealed into a token that is provided to all other nodes during (re)deployment. To join a swarm, a node verifies the identity of the remote manager based on the token, generates a preliminary certificate data and sends a certificate signing request (CSR) to the manager together with the token. After verifying the secret from the token, the manager issues a certificate to the node. When a node needs to connect to another node, both nodes will authenticate to each other and set up a TLS tunnel, using the certificates issued to them by the CA. See Figure 4.3.

The default behavior for the nodes is to automatically renew their certificate every three months, but shorter intervals can also be used. The update does not happen simultaneously for all the nodes but instead takes place within given time-frame due to security reasons. Docker Swarm also supports rotation of the CA certificate and embedding into already existing PKI. Another popular container orchestration solution, Kubernetes, does not support MTLS with automated certificate provisioning at the moment, but the work on Kubelet TLS Bootstrap feature is ongoing.

Netflix Case

Netflix internal microservice network utilizes a PKI based on short-lived certificates for TLS with mutual authentication [101]. The Netflix approach builds on the notions of short- and long-lived credentials. The long-lived credentials are provisioned into the service during a bootstrap procedure, stored either in Trusted Platform Module (TPM) or SGX, and are required to obtain and update short-lived credentials. While the Docker source code is publicly available, the Netflix PKI solution is not.

The idea of issuing certificates with a short lifetime as a solution to the certificate revocation problem on the Web has been suggested before [109]. The short expiration time of certificates limits the utility for revocation mechanisms.

TLS with mutual authentication addresses problems of service authentication and traffic encryption, but not service authorization. Moreover, user to service authentication and authorization are still left to the discretion of developers.

4.5.2 Principal Propagation via Security Tokens

After a user has been authenticated by the gateway, the microservices behind it will be processing user's requests. Microservices should be aware of the user authentication state, i.e. whether the user was authenticated, and what the user's role is in authorization context. The user needs to be identified multiple times in each service down the operation chain, as each service calls other services on the user's behalf.

Security tokens and relevant standards

Token-based authentication is a well known commonplace security mechanism that relies on cryptographic objects called security tokens containing authentication or authorization information. A security token is created on the server side upon the successful validation of the client's credentials and given to the client for subsequent use. Security tokens substitute the client's credentials within limited time-

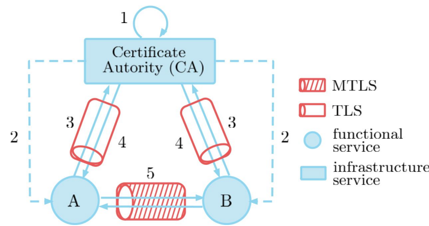


FIGURE 4.3: *A generic solution for microservice trust based on MTLS. 1) CA service generates a self-signed root certificate and a shared secret. 2) A cryptographic hash of the certificate and a shared secret are extracted from the CA service and provisioned into the new benign service either automatically or manually. 3) A benign service establishes a one-way TLS connection with the CA service, verifies CA identity using the provisioned hash of the CA certificate, and if successful submits a CSR and shared secret. 4) Upon successful verification of the received CSR and shared secret, CA service signs and sends back the newly issued certificate. 5) Two benign services communicate over MTLS.*

frame. Token-based authentication via HTTP cookies is a prominent example. Fu et al. [39] gives a detailed security evaluation of the approach.

Token-based authentication advanced even further with the widespread adoption of OpenID Connect [112], a security standard for single sign-on and identity provisioning on the Web. The same functionality is provided by the Security Token Service (STS). STS is a component of the WS-Trust standard [83] that extends WS-Security standard with methods for issuing, renewing, and validating security tokens. Other relevant standards are JSON security standards: JSON Web Signature (JWS), Encryption (JWE), and Token (JWT).

Reverse Security Token Service

A noteworthy trend in the industry is the use of JWT for principal propagation within the microservice network. Multiple informal records of the approach can be found [25]. Although no formal description of it exists in scientific literature, those familiar with the above-listed security standards will find the suggested approach closely related to the existing standards.

Token-based user-to-service authentication where each service understands tokens allows transporting user identity and user session state through the system in a secure and decentralized manner. After the user is authenticated with the authentication service, a security token representing the user will be generated for *internal* use within the microservice network. In this paper, we refer to a separate service that is responsible for the token generation as a Reverse STS.

Limited lifetime of the security token is achieved by including an expiration time in its body. For security reasons, a shorter token lifetime is desired. Information about the user and intended audience can be included if needed. The token will be passed to the microservices involved in processing the request. Before executing the received request each microservice will validate the adjacent token using a corresponding public key. Token validation is a mandatory first step of the request

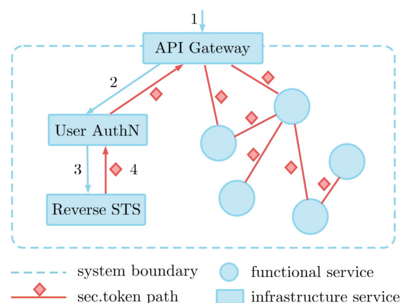


FIGURE 4.4: *A generic token-based authentication scheme for microservices that enables a user-to-service authentication and identity propagation based on cryptographic tokens. 1) Incoming user request hits API Gateway. 2) APIGateway prompts user for authentication by redirecting to the dedicated user authentication service. 3) Requesting a security token. If user is authenticated successfully, the authentication token service generates a token that represents the given user inside the system. 4) Returning a security token. The token is passed alongside the user request to the downstream services. This token is designated for internal use only and is not given to the user.*

processing. See Figure 4.4.

This approach fits well with the second design principle of distributed systems: individual nodes make decisions based on locally available information (see Section 4.2.4). It facilitates loose coupling of the services and is highly scalable while having no overhead of a centralized solution. Moreover, there is good tool support for this approach. OAuth 2.0, a standard for delegated authorization, and OpenID Connect, an authentication layer on top of OAuth 2.0, can be tailored for inter-service security.

There are three caveats with this approach. The first is an assumption that the clock synchronization problem is non-existent. To validate tokens, both the token issuing node and the node performing validation must have their clocks synchronized. This is usually straightforward to handle, e.g., using NTP. The second one is that the tokens must be sent over a protected channel, i.e. TLS, otherwise the tokens can be intercepted and re-used within their validity period. Having short validity time is a partial solution to this problem. The third one is that the private key of the token issuing service must be kept safe at all times. If the private key is compromised, any user can be impersonated by the attacker.

4.5.3 Fine-Grained Authorization

Although no prominent trends for microservice authorization exist in industry at the moment, we will mention several promising approaches.

Security Tokens for User Authorization

Various access control mechanisms exist including Role Based Access Control (RBAC) and Attribute Based Access Control (ABAC). RBAC and its predecessors are user-centric access control models. Therefore, they do not account for relation-

ship between the requesting entity and the resource. For fine-grained authorization on resources, such as access to a specific API call, ABAC should be used.

Security tokens can include authorization information. For example, RBAC authorization roles can be incorporated into JWT tokens as an additional attribute.

Inter-Service Authorization Based on Certificates

Use of digital certificates for authorization rather than authentication was first suggested in 1999, e.g. Simple Distributed Security Infrastructure (SDSI) [110] and Simple Public Key Infrastructure (SPKI) [27]. Later, Marcon [69] proposed a certificate-based permission management system for SOA.

Not all microservices are deployed equal: only microservices co-dependent by design should be able to call each other. If the microservice network already relies on MTLS and self-hosted PKI, the same PKI can provide the basic service to service authorization. A separate signing certificate should be created per microservice *type*. This certificate will be used to sign the certificates of all instances of the same type.

Let's assume there are three types of microservices: A, B, and C. There are multiple instances of each type. While B is connected to both A and C, no direct connection is allowed between A and C. The default rule is to trust no one. To allow access to B from A and C, all instances of B should be preconfigured to trust the certificates signed by certificate type A and C. To allow access to A and C from B, the certificate for type B should be added to A's and C's trust lists.

4.6 Microservice Security Framework

As shown, no standard way to deal with microservice security concerns exists. The existing implementations are often closed source (Netflix MTLS), not directly portable to other environments (Docker Swarm MTLS), and in general not well documented or understood. Moreover, a performance cost of using the existing security solutions in a microservice setting is unknown. To partially address this problem we implemented a microservice security framework, *MiSSFire* that provides a standard way to embed security mechanisms into microservices. Furthermore, we evaluated the performance of the framework against a toy microservice-based bank system of our design *(MicroBank)*.

4.6.1 Design and Implementation

When designing the MiSSFire framework we tried to address the main microservice security challenge—the problem of establishing trust between individual microservices. The core design criteria were security, scalability, and automation. We followed the defense in depth principle and the attack model introduced in Section 4.3.2. Security mechanisms that we implemented are heavily based on the emerging security practices from Section 4.5, specifically mutual authentication of services using MTLS and principal propagation via JWT.

The framework consists of a set of infrastructure services that need to be up and running within a system and a template for a regular functional service. Cur-

rently, the framework is bundled with two infrastructure services that expose relevant REST APIs:

- *The CA service* is a core part of the self-hosted PKI that enables MTLS between microservices. It generates a self-signed root certificate and signs CSR from other services. See Figure 4.3 for more details.

- *The Reverse STS* stays behind a user authentication service (not included) and generates security tokens in JWT format. A new JWT is generated per user request. See Figure 4.4 for more details.

The template for a regular functional service simplifies integration with the infrastructure services by providing all the necessary functionality. Additionally, it forces use of MTLS for all connections and requires presence of JWT for all incoming requests.

4.6.2 Experiment

Although performance is not a security property, it has been an important deciding factor for adoption or rejection of security mechanisms in the real world. In this section we address the question of how the common security mechanisms that are bundled in our framework impact the performance of an actual microservice-based system.

MicroBank

To test the framework we needed an actual microservice application. For this purpose we developed our own fictitious microservice-based bank system that consists of the following microservices:

- *API gateway*: The main entry point to the system.

- *Accounts*: Manages user accounts.

- *Payment*: Provides payment functionality.

- *Transactions*: Handles transaction operations.

- *Users*: Manages users.

The system was built using mainstream development techniques for microservices. The system is written in the Python v2.7 programming language. Although the microservice style does not dictate what communication protocols or styles should be used, in practice REST APIs and JSON format are the default choices. Each service in the system exposes a set of relevant REST APIs. This is done by using the web framework Flask and WSGI HTTP server Gunicorn. The number of workers per server is adjustable.

The 'shared nothing' architecture is a central part of the loose coupling concept in microservice world. Therefore, Accounts, Transactions, and Users microservices

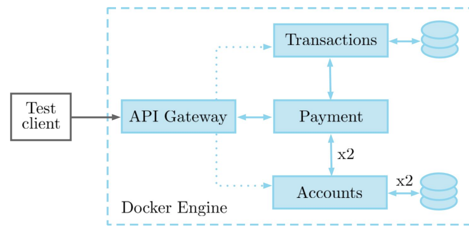


FIGURE 4.5: *Experiment setup: payment operation.*

maintain individual SQLite databases. For the simplicity of the experiment there is only one instance of each microservice type.

The services of the bank model can be run as separate processes or in Docker containers. The whole bank model can also be run as a multi-container Docker application using the compose tool (though both the framework and the sample application can work independently of Docker). The tool automates configuration and creation of containers and allows to start the whole system with a single command. The source code has a partial unit-test coverage.

Methodology

The test client registers two users and opens bank accounts for them. Then, the test client carries out a series of payment operations between the two users. In the experiment, 50 concurrent test clients are started simultaneously, where each test client performs 100 sequential payment operations. To measure the system performance, an average execution time of 5000 payment requests is calculated on the client side.

The payment operation involves four microservices as shown in Figure 4.5. Several factors contribute to the time it takes to perform one payment operation. These factors are network delays and processing time inside microservices including database access time.

In the given setup, payment operations always succeed. If the operation is invalid, such as an attempt of transferring a negative amount, or if the recipient bank account is closed in the middle of the payment operation, the implemented system will roll back to revert the partially made changes. This would adversely affect the payment operation execution time.

The experiment consists of four parts:

- *Baseline.* Running the test client against the bank model with security features disabled.
- *Tokens.* Running the test client against the bank model with the Reverse STS service in place and JWT tokens validation.
- *MTLS.* Running the test client against the bank model with the CA service in place and all communication secured with MTLS.

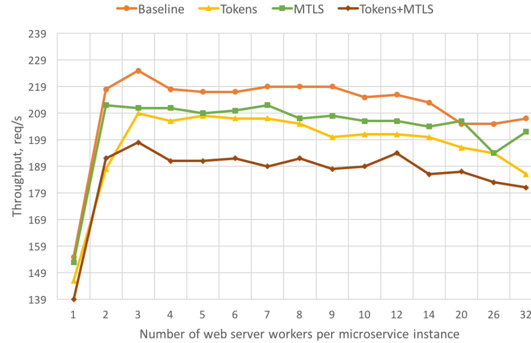


FIGURE 4.6: *Performance of the bank model under load of 50 test clients making payments.*

- *Tokens+MTLS.* Running the test client against the bank model with all security features enabled.

The experiment was run on a MacBook Pro with a 2,6 GHz Intel Core i7 processor and 16 GiB of memory. The computing resources dedicated to Docker were 8 CPUs and 8 GiB of memory. Both the test clients and the bank model resided on the same machine, all communication was performed via localhost.

Results and Discussion

Figure 4.6 presents the results of our experiment. As expected the performance in the baseline case is higher than in the cases of tokens and MTLS. Closer inspection of the data shows that the tokens decrease performance by 7% on average, while the MTLS impact is around 4%. The most interesting aspect of this figure is that the difference between the baseline case and the case with all security mechanisms in place is relatively small, and accounts for around 11% in the given setup. Based on the fact that microservice solutions are slow in general, we believe this security overhead is still acceptable, especially for security critical applications. There is a relatively high cost to setting up HTTPS connections and validating tokens, which we reduce by pooling and reusing existing connections between services when possible.

Ueda et.al. [128] showed that the performance of the microservice version can be around 80% lower than the monolithic version on the same hardware configuration. This is a significant overhead that industry is willing to take and in many cases has already embraced. This also means that the overhead of having a microservice-based solution in first place is so high, that the impact of security mechanisms becomes insignificant in comparison.

From the chart, it can be seen that the best throughput is achieved with 3 workers (except for the MTLS case). When more workers are added the throughput starts to deteriorate slowly. These results deviate from an expected behavior where the throughput scales linearly up to the number of cores available, assuming the parallelizable portion of the program completely dominates the execution time. This

may be related to the clients sharing the same CPU; though it may also be due to specifics of Gunicorn.

The relatively slow performance of our bank model can be attributed to the following: 1) Synchronous HTTP communication: sequential requests and no parallelization; 2) Slow database access. 3) Python is not as efficient as certain other programming languages; 4) Suboptimal configuration of the web servers. Also, the bank model is only a proof-of-concept and offers limited performance and scalability.

4.6.3 Evaluation

Security Considerations

Our framework relies exclusively on well-known security mechanisms and standards such as SSL and JWT. No new cryptographic primitives or protocols are introduced. Although implementation flaws are always possible, we need to rely on security and trustworthiness of the building blocks.

The CA service and the Reverse STS are security-critical. It makes sense to run these two services in a hardened environment. SGX-capable servers in the public cloud offered by Microsoft Azure [76] is one of possible solutions.

Performance Evaluation

Our experiments show that a microservice network introduces latencies on the order of milliseconds; in this setting the performance hit of basic security features becomes negligible.

Framework Limitations

Currently, the framework lacks multiple important security features. It is not a production-ready tool, and should mostly be seen as a proof of concept. The framework is overly simplified: it does not address authorization, there is no key rotation or key revocation mechanisms. Also, it is currently limited to Python.

Reproducibility and Future Work

To support reproducible research and allow others to improve on our results we released our code open source under GNU GPLv3 license. The source code of both MiSSFire framework and the MicroBank are publicly available at GitHub (https://github.com/yarygina/MiSSFire and MicroBank), including setup and benchmarking scripts. In future investigations, we intend to improve on the aforementioned limitations of our framework.

4.7 Conclusion

In this paper, we have examined the microservice architectural style, with a particular focus on its security implications. Microservices bring together concepts from both service-orientation, distributed systems and fundamental software engineering principles of abstraction, reuse and separation of concerns. This combination

brings both new challenges that must be addressed, as well as old security challenges in a new wrapping. The microservice style of highly isolated, easily redeployable distributed components also implies new opportunities for better security, e.g., through increased diversity or restricting data access to only the services that need it.

As a concrete example of microservice-specific security, we have developed a small, openly available prototype framework for establishing trust and securing microservice communication with MTLS, self-hosted PKI and security tokens. Our case study shows that there is little extra performance cost to securing microservice communication, likely due to the overall high overhead of the communication itself.

With increased industry adoption of microservices, and the overall increasing threat level on the Internet, researching and developing secure microservices is crucial; and, with microservices seen as a lightweight, easy-to-use approach to SOA, we believe it is particularly important that security solutions are also lightweight, easy to use, and accessible to real-world developers.

References

[1] N. Alshuqayran, N. Ali, and R. Evans.
"A Systematic Mapping Study in Microservice Architecture".
In: *Service-Oriented Computing and Applications (SOCA 2016)*. Nov. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15.

[4] J. Armstrong.
"Making Reliable Distributed Systems in the Presence of Software Errors".
PhD thesis. Stockholm, Sweden: The Royal Institute of Technology, Dec. 2003.

[14] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull.
"Exploring Language Support for Immutability".
In: *Int'l Conf. on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 736–747. DOI: 10.1145/2884781.2884798.

[15] T. Combe, A. Martin, and R. Di Pietro.
"To Docker or not to Docker: A security perspective".
In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[25] B. Doerrfeld. *How To Control User Identity Within Microservices*.
Nordic APIs blog, 2016.
URL: https://nordicapis.com/how-to-control-user-identity-within-microservices/ (visited on Nov. 1, 2017).

[26] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. "Microservices: yesterday, today, and tomorrow".
In: *CoRR* abs/1606.04036 (2016).

[27] C. M. Ellison, B. Frantz, B. Thomas, T. Ylonen, R. L. Rivest, and B. Lampson.
SPKI certificate theory. IETF RFC 2693, 1999.
URL: https://tools.ietf.org/html/rfc2693 (visited on Nov. 1, 2017).

[32] C. Fetzer. “Building Critical Applications Using Microservices”. In: *IEEE Security Privacy* 14.6 (Nov. 2016), pp. 86–89. ISSN: 1540-7993. DOI: 10.1109/MSP.2016.129.

[38] M. Fowler and J. Lewis. *Microservices*. Mar. 2014. URL: http://www.martinfowler.com/articles/microservices.html (visited on Nov. 1, 2017).

[39] K. Fu, E. Sit, K. Smith, and N. Feamster. “The Dos and Don’ts of Client Authentication on the Web.” In: *USENIX Security Symposium*. 2001, pp. 251–268.

[57] N. Josuttis. *SOA in Practice: The Art of Distributed System Design*. Sebastopol: O’Reilly Media, Inc., 2007. ISBN: 0596529554.

[59] S. Kim, F. B. Bastani, I.-L. Yen, and R. Chen. “High-assurance synthesis of security services from basic microservices”. In: *Software Reliability Engineering (ISSRE 2003)*. IEEE. 2003, pp. 154–165.

[60] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01203.

[68] A. Madhavapeddy and D. J. Scott. “Unikernels: Rise of the Virtual Library Operating System”. In: *Queue* 11.11 (Dec. 2013), 30:30–30:44. ISSN: 1542-7730. DOI: 10.1145/2557963.2566628.

[69] A. L. Marcon, A. O. Santin, L. A. de Paula Lima, R. R. Obelheiro, and M. Stihler. “Policy control management for Web Services”. In: *Integrated Network Management (IM’09)*. IEEE. 2009, pp. 49–56.

[70] V. Mavroudis, A. Cerulli, P. Svenda, D. Cvrcek, D. Klinec, and G. Danezis. “A Touch of Evil: High-Assurance Cryptographic Hardware from Untrusted Components”. In: *Computer and Communications Security (CCS’2017)*. ACM, 2017, pp. 1583–1600. DOI: 10.1145/3133956.3133961.

[76] Microsoft Azure. *Introducing Azure confidential computing*. Sept. 2017. URL: https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/ (visited on Nov. 1, 2017).

[78] B. Möller, T. Duong, and K. Kotowicz. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. Sept. 2014. URL: https://www.openssl.org/~bodo/ssl-poodle.pdf (visited on Nov. 1, 2017).

[79] D. Mónica. *MTLS in a Microservices World*. Talk at BSides Lisbon 2016. Nov. 2016. URL: https://www.youtube.com/watch?v=apma_C24W58 (visited on July 1, 2018).

[80] F. Montesi and J. Weber. “Circuit Breakers, Discovery, and API Gateways in Microservices”. In: *CoRR* abs/1609.05830v2 (2016). URL: http://arxiv.org/abs/1609.05830v2.

[83] A. Nadalin, M. Goodner, M. Gudgin, D. Turner, A. Barbir, and H. Granqvist. *OASIS Standard Specification. WS-Trust 1.4*. Apr. 2012. URL: http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html (visited on Nov. 1, 2017).

[85] S. Newman. *Building Microservices*. O'Reilly Media, 2015. ISBN: 9781491950357.

[90] *Official Docker v17.06 documentation. Manage swarm security with public key infrastructure*. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki (visited on Nov. 1, 2017).

[93] C. Otterstad and T. Yarygina. "Low-Level Exploitation Mitigation by Diverse Microservices". In: *Service-Oriented and Cloud Computing (ESOCC 2017)*. Ed. by F. De Paoli, S. Schulte, and E. Broch Johnsen. Oslo, Norway: Springer, 2017, pp. 49–56. DOI: 10.1007/978-3-319-67262-5_4.

[94] *OWASP Top 10 – 2017: The Ten Most Critical Web Application Security Risks*. 2017. URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%5C%28en%5C%29.pdf.pdf (visited on Nov. 1, 2017).

[98] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. "Microservices in Practice, Part 1: Reality Check and Service Design". In: *IEEE Software* 34.1 (Jan.–Feb. 2017), pp. 91–98. DOI: 10.1109/MS.2017.24.

[99] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. "Microservices in Practice, Part 2: Service Integration and Sustainability". In: *IEEE Software* 34.2 (Mar.–Apr. 2017), pp. 97–104. ISSN: 0740-7459. DOI: 10.1109/MS.2017.56.

[100] C. Pautasso, O. Zimmermann, and F. Leymann. "RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision". In: *Proceedings of the 17th International Conference on World Wide Web (WWW'09)*. Beijing, China, 2008, pp. 805–814. DOI: 10.1145/1367497.1367606.

[101] B. Payne. *PKI at Scale Using Short-lived Certificates*. Talk at USENIX Enigma 2016. URL: https://youtu.be/7YPlsbz8Pig (visited on July 1, 2018).

[103] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. "Flip Feng Shui: Hammering a Needle in the Software Stack". In: *25th USENIX Security Symposium*. USENIX, Aug. 2016, pp. 1–18. ISBN: 978-1-931971-32-4.

[106] M. Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, 2015. ISBN: 978-1-491-94161-4.

[107] C. Richardson and F. Smith. *Microservices: From Design to Deployment*. NGINX, Inc., 2016. URL: https://www.nginx.com/resources/library/designing-deploying-microservices/.

[109] R. L. Rivest. “Can We Eliminate Certificate Revocation Lists?” In: *In Financial Cryptography*. Springer-Verlag, 1998, pp. 178–183.

[110] R. L. Rivest and B. Lampson. *SDSI—a simple distributed security infrastructure*. MIT. 1996. URL: https://people.csail.mit.edu/rivest/sdsi10.html.

[112] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. *OpenID Connect Core 1.0*. 2014.

[123] Y. Sun, S. Nanda, and T. Jaeger. “Security-as-a-service for microservices-based cloud applications”. In: *Cloud Computing Technology and Science (CloudCom 2015)*. IEEE. 2015, pp. 50–57.

[124] H. Takabi, J. B. D. Joshi, and G. J. Ahn. “Security and Privacy Challenges in Cloud Computing Environments”. In: *IEEE Security Privacy* 8.6 (Nov. 2010), pp. 24–31. ISSN: 1540-7993. DOI: 10.1109/MSP.2010.186.

[125] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007. ISBN: 9780132392273.

[126] *The Heartbleed bug in OpenSSL library*. Apr. 2014. URL: http://heartbleed.com/ (visited on Nov. 1, 2017).

[128] T. Ueda, T. Nakaike, and M. Ohara. “Workload characterization for microservices”. In: *IISWC 2016*. IEEE. 2016, pp. 85–94.

[133] Y. Yarom and K. Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *23rd USENIX Security Symposium*. USENIX, 2014, pp. 719–732. ISBN: 978-1-931971-15-7.

[134] T. Yarygina. “RESTful Is Not Secure”. In: *Applications and Techniques in Information Security (ATIS 2017)*. Springer, 2017, pp. 141–153. DOI: 10.1007/978-981-10-5421-1_12.

[138] O. Zimmermann. “Microservices tenets: Agile approach to service development and deployment”. In: *Computer Science - Research and Development* (2016), pp. 1–10. ISSN: 1865-2042. DOI: 10.1007/s00450-016-0337-0.

Paper II

Low-level Exploitation Mitigation by Diverse Microservices

Christian Otterstad, Tetiana Yarygina

5

This paper is an extended version of C. Otterstad and T. Yarygina. "Low-Level Exploitation Mitigation by Diverse Microservices". In: *Service-Oriented and Cloud Computing (ESOCC 2017)*. Ed. by F. De Paoli, S. Schulte, and E. Broch Johnsen. Oslo, Norway: Springer, 2017, pp. 49–56. DOI: 10.1007/978-3-319-67262-5_4.
Original paper is © 2017 Springer; reused with permission.

Low-level Exploitation Mitigation by Diverse Microservices

Christian Otterstad, Tetiana Yarygina

Department of Informatics, University of Bergen, Norway

Abstract This paper discusses a combination of isolatable microservices and software diversity as a mitigation technique against low-level exploitation. The effectiveness and benefits of such an architecture are substantiated. We argue that the core security benefit of microservices with diversity is increased control flow isolation. A simple implementation of a microservice network is given as a proof of concept of the added isolation of the control flow. Exploitation attempts are made against the microservice network and a monolithic counterpart, and the results are discussed to support the assertion. Finally, a new microservices design pattern leveraging a security monitor service and anti-fragility to low-level exploitation is introduced to further utilize the architectural benefits inherent to microservice architectures.

Keywords *security, software diversity, design patterns, robustness, anti-fragility*

5.1 Introduction

Microservices is a recent trend in software design. A microservice architecture simplifies the development of complex horizontally scalable systems that are highly flexible, modular, and language-agnostic. These factors contribute to the increasing popularity of microservices both in industry and academia. According to survey results from NGINX [86], one in three IT companies had microservices in production as of late 2015, and even more were planning to start using microservices. Numerous sources, including books [85, 107], research papers [98], and various online sources [37], discuss advantages and disadvantages of microservice solutions.

We define a microservice as a small specialized autonomous service communicating over a network boundary. By extension, a microservice system is a distributed software system consisting of a set of microservices communicating to perform

some computation as an aggregated result of their collective operation. Similarly to how a computer program is typically divided into procedures, the whole system is divided into individual services. For further information, we refer the reader to the comprehensive study of microservice principles by Zimmermann [138] who identified commonalities in the popular microservice definitions and concluded that microservices represent a development- and deployment-level variant of the service-oriented architecture (SOA).

Although microservice architectures constitute an important trend in software design with major implications in software engineering, surveys such as the one conducted by Dragoni et al. [26] have highlighted a general lack of research in the area of microservice security. In Newman's book [85] on microservice design, a subset of security traits for improving the security of microservice networks is discussed. The idea of combining microservices with secure containers and compiler extensions to build critical software has been investigated in a recent study by Fetzer [32]. The paper by Lysne et al. [67] briefly introduces the notion of microservice networks to mitigate vendor-malware and other forms of attacks, without any further elaboration or working examples.

Herein, we expand and elaborate on the generalized notion of mitigating low-level exploitation. To our knowledge, we are the first to demonstrate the benefits of using a microservice architecture to defend against remote low-level exploitation. Unlike a deployment monolith, a microservice architecture facilitates strong process isolation partly because the services run on different physical machines or in different virtual environments. The advantages of process isolation are demonstrated by carrying out low-level attacks on a simplified bank application, implemented as both a monolith and a microservice solution. The paper also introduces a security monitor service that further leverages the architectural benefits of a microservice network, including added software diversity, to enable anti-fragility to low-level exploitation.

The rest of the paper is organized as follows. In Section 5.2, the attack model is presented and discussed. The same section also introduces the various attack vectors and the basic types of exploits that are topical to this paper. Section 5.3 provides key design rules for microservice networks in the context of security. Section 5.4 introduces a security monitor system to respond to security-related incidents. Section 5.5 contains the programming example demonstrating that microservice solutions provide added protection against low-level exploits compared to deployment monoliths. Finally, Section 5.6 concludes the paper.

5.2 Model Overview and Exploitation Analysis

This section introduces a model of a microservice solution and discusses the basic exploitation primitives that we assume are available to an attacker. Later, this model and its exploitation primitives will be used to demonstrate the security benefits of microservice solutions compared to deployment monoliths.

5.2.1 Model Overview

The general model applicable to this study is a microservice network where we consider generic functionality offered to external users. There are several possible designs for how such a network can be structured and hosted. As they are all applicable in this context, they will be briefly described. The common trait to all of the designs is control flow isolation through some mechanism. This mechanism can be—in the order of increasing strength—traditional process isolation, containers, a hypervisor, or physical machine isolation. The schemes offering stronger isolation tend to be more costly for the defender in terms of overhead and additional hardware as compared to the less strongly isolated configurations.

Traditional process isolation exposes the whole user space kernel interface to each process. Containers enforce stronger isolation, whereby only a subset of the user space kernel interface is accessible to the process. Hypervisor isolation enables the processes to reside in a virtual machine, where the interface exposed is mostly limited to the hypervisor itself. Physical machine separation completely removes the dependency on the same physical hardware as two processes run on different machines. The reason the stronger types of isolation are desirable will be justified later in the paper. The microservices communicate in the same manner in all cases.

5.2.2 Exploitation Overview

In general, an attacker wants to gain access to an asset controlled by a defender, extending up to full access to the targeted system where root shell access or equivalent is typically the most desired.

The system offers the attacker, as well as any ordinary user, some form of functionality, such as a web store, which allows any user to put items in the shopping basket or make a purchase. While this functionality is intended by the developer, the attacker's goal is to extend the set of possible operations beyond the intended functionality. The goal is achieved through exploitation. For low-level exploitation, the attacker often takes control over the program by hijacking the control flow of execution, with program execution often facilitated by code-reuse techniques such as return-oriented programming [116]. While high-level exploitation, which does not directly take control over the program counter, is also possible, we are only concerned with low-level exploitation in this paper.

It is assumed that the external attacker is able to carry out the following types of exploits: an initial exploit (E_{init}), a virtual machine or sandbox escape exploit (E_{VM}), and a lateral exploit (E_{lat}). E_{init} is used to gain a shell on a microservice node, E_{VM} enables the attacker to escape from a sandbox, while E_{lat} is an exploit type that abuses the trusted relationship between microservice nodes in cases where additional attack surface is needed and E_{init} is not sufficient.

Figure 5.1 illustrates an attack propagation through a microservice network with diverse microservices running in virtualized environments on networked machines. The attacker initially obtains access using E_{init} and then proceeds to escape the sandbox using E_{VM}. Once the attacker has executed the latter exploit, full control over all

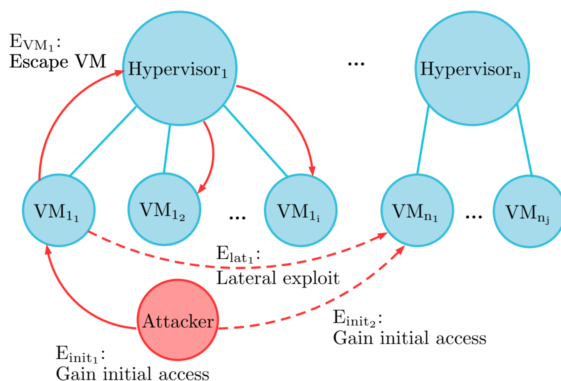


FIGURE 5.1: *Attacking a microservice architecture with diverse microservices running in virtualized environments on networked machines.*

nodes hosted by the same hypervisor is obtained. However, the attacker does not control the whole network. To extend the control further, the process must basically be repeated. However, the same exploit E_{init_1} may not work against VM_{n_1}—a node hosted by a different machine n, which cannot be reached through the hypervisor. Therefore, the attacker will have to resort to either using a different exploit E_{init_2}, or, depending on the available attack surface and overall exploitability, a lateral exploit E_{lat_1} to utilize the now exposed trusted relationship between the nodes.

There exist various techniques to mitigate low-level exploits. Examples are virtualization techniques (e.g. VT-x), ASLR (Address Space Layout Randomization), NX-bit (No eXecute), canaries, MPX (Memory Protection Extensions), SGX (Software Guard Extensions), SMAP (Supervisor Mode Access Prevention), IDS (Intrusion Detection System), and XnR (Execute no Read).

5.3 Microservice Architecture and Its Security Merits

Independently of whether a new microservice-based system will be built from scratch or an existing monolithic system will be transformed into a microservice network, several important architectural decisions must be made. This section discusses the security benefits of microservice architectures as well as how the common microservice design patterns affect security. Additionally, this section elaborates on the notion of robustness (hardening) and how to prevent an attacker from spreading between microservices as first presented in the Lysne et al. paper [67].

5.3.1 Microservice Design Patterns Affecting Security

Before discussing microservice architectures in a security context, we outline a few basic design patterns. The literature presents various design patterns that a microservice oriented system might employ. Although we focus on a microservice architec-

ture, many design patterns originate from the world of distributed systems preceding microservices.

- *API Gateway* [80, 107] is the entry point for all clients. A system without an API Gateway or equivalent would need to expose the required services to external users—hence increasing the initial attack surface. From a security perspective, the API Gateway provides an additional obstacle for the attacker in the sense that the API Gateway must likely be compromised in order to expose the internal services, assuming the internal services only trust the API Gateway by design.

- *Service Discovery* [80, 107] is a centralized scheme allowing services to discover other services. It is used because manual updates are infeasible in practice. Alternatively, a distributed peer-to-peer system could be used to exchange lists of available nodes, although likely at the cost of increased complexity. An attacker can exploit the service discovery to determine the internal structure and communication patterns between services. If the attacker manages to compromise the service discovery, then the attacker might be able to host malicious services and redirect traffic to them when their addresses are requested by benign services, thus exposing a client-side attack surface on the services being targeted.

- *Circuit breaker* prevents cascading failures by changing the component behavior based on the number of failed calls made. The pattern was popularized in the book by Nygard [89], and has received significant attention since [80]. The Netflix Hystrix library provides an implementation of the pattern.

5.3.2 Security Considerations

There are two distinct types of microservices in the context of interaction: microservices that allow both external and internal interaction and microservices that only allow internal interaction. Internal interaction is communication between two microservices within the system boundary. External interaction is interaction between an external host and a microservice that is part of the system. A microservice that only allows external interaction is effectively defined as a monolithic program.

However, regardless of the type of microservice and of the granularity at which microservices are implemented, every microservice must contain functionality for network interaction. The code the user can directly interact with is the most obvious attack vector. The microservices must assume that any input encountered is hostile. Not only are the microservices communicating over an insecure network, but some of the nodes in the network may be compromised. Therefore, even properly authenticated nodes should not trust the subsequent input to be sane or properly formatted by its peer(s).

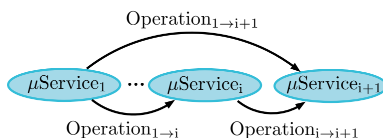


Figure 5.2: *Depiction of an unnecessary edge, exposing additional attack surface.*

A *robust system* is basically what is commonly referred to as a hardened system. Robustness is a property we use to denote how much effort is required to successfully perform a low-level exploit against the system. The following discussion covers some security considerations specific to enhancing the robustness of microservice networks.

Maximizing API security. Exposed network interfaces must be minimal, have strong input validation, and be of the highest type in the Chomsky hierarchy [113]. These are well-known design traits for a secure system, and they apply equally to both monolithic designs and microservice designs. If there is any way to accomplish the same functionality while exposing the server to less computation on external input, this is advisable. The defender should strive to minimize the set and depth of possible control flow paths that the attacker can influence at any step.

Avoiding unnecessary node relationships. The defender must employ an architecture that prevents unnecessary node relationships. Consider Figure 5.2. If μService$_1$ can reach μService$_{i+1}$ through μService$_i$, then there should not be any edge between μService$_1$ and μService$_{i+1}$. Adding the extra edge may increase the attack surface for the involved nodes. While taking a shortcut of this type to obtain information or perform functions directly might result in better performance and less complexity, doing so would violate the trade-off of increased security for less performance and higher complexity. If a microservice network forms a dense graph, then most likely the design of such a system and/or its decomposition into microservices is incorrect.

Asymmetric node strength. To optimize the robustness of the network to low-level exploitation, the more secure nodes should be placed at critical network segments, such as entry points and nodes guarding the more valuable assets, as shown in Figure 5.3. A more priced asset could be functionality that allows making a transaction as compared to merely viewing the list of already performed transactions. The payment functionality could use most of the budget for hardening whereas viewing an account is considered less severe and should not be as prioritized. Examples of hardening are given in the next section. High diversity as a mechanism for hardening microservices is also discussed in the next section. Such changes can be done a priori, in contrast to tactical choices based on real world statistics.

5.3.3 Security Through Diversity

The purpose of diversity is to make an exploit less statistically likely to succeed and to make the attack scale less effectively, thus, providing the defender with time to react to the attack. The most common (as of 2017) examples of diversity in computer systems are the use of different programming languages, hardware architectures, cloud providers, operating systems, hypervisors, compilers or compiler arguments, and ASLR versions [19, 47, 132] that enable identical programs to possess diversity.

It should be stressed again that a microservice system has inherent diversity, simply as a consequence of microservices implementing different functionality. Different bugs are assumed to be associated with different functionality. However, this may not be true in all cases—two microservices with different functionality could employ a common library with an exploitable vulnerability. It should also be pointed out that while some type of diversity *may* alter the nature of a bug, a successful replay attack using the same exploit may still be possible.

A defender should make a system with as much diversity as possible. Minimal diversity has previously been defined [96] as "when failure of one of the versions is always accompanied by failure of the other". This definition is also applicable in the context of exploitation. If there is so little diversity that the exact same exploit works equally well on both versions, then the diversity is of no benefit to the defender. However, it should be stressed that the diversity still serves a purpose in terms of redundancy against other types of failures, but not against targeted attacks.

N-VERSION PROGRAMMING. The expansion of one node into multiple nodes through N-version programming allows the set of nodes to be more robust due to their inherent diversity as compared to a single node. In the N-version programming scheme, a voting system is employed where a majority must be reached by a set of nodes performing the same computation in parallel. Therefore, being able to exploit a subset of nodes less than the limit used by the voting system to make choices would not give the attacker the same control as if no N-version programming was used.

In the special case where the attacker controls exactly half, the attack is reduced to a denial of service attack, as the defender can choose to ignore all the input. Any

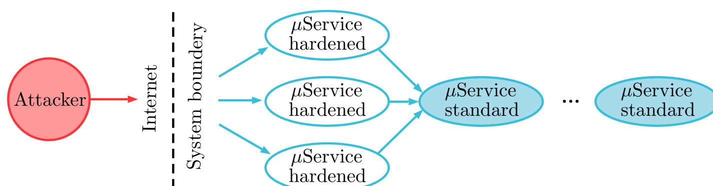


FIGURE 5.3: *The use of asymmetric node strength to defend against low-level attacks.*

Algorithm 1 The Security Monitor basic operation

```
1: procedure MutateService
2:     if IDS() is true: then
3:         Kill the service environment.
4:         Diversify.
5:         Rebuild the environment and restart the service.
```

node can be dynamically expanded in this manner, and any set of expanded nodes can be dynamically reduced back to a smaller number of nodes or a single node. This expansion and contraction can be performed automatically based on the available resources to the system.

5.4 The Security Monitor Service

Normally, a system will only get patched after developers have identified issues and rolled out the changes. Although this improves the system over time it can introduce a large attack window due to the inherent latency of the process. A microservice network may automate some of the issues that arise, specifically by introducing a security monitor system. The security monitor can identify nodes that either report erroneous data, trigger IDS detections, or simply report inconsistent data compared to their siblings in an N-version programmed subsystem. Anomalous behavior may result in the monitor taking explicit, autonomous action. The goal of the monitor would always be to remove the attacker from the system, as well as introduce diversity into the system in order to make it less likely that the same attack will succeed if attempted again.

5.4.1 Design Overview

The security monitor has at its disposal an arbitrary set of ways to introduce new diversity. This set would likely be largely dictated by the budget of the defender, but could consist of different N-version programmed versions of the same service, strong ASLR implementations, earlier versions of the service, and different versions of libraries. The controller can decide to employ some or all of these at the diversification step in the case that a security issue is detected. The overall operation of the service monitor is depicted in Algorithm 1.

N-version programming with microservices. A simple example would be an N-version programmed system with a set of nodes that perform the same task using compiler derived diversity [54]. Similarly to the N-variant system suggested by Cox et al., we propose a scheme to exploit the fact that the defender retains part of the control flow of the overall system [17]. For security critical systems, individual microservices can be implemented as N-version programmed systems. The nodes within such a system will perform the same task using compiler derived diversity [54]. If a particular node issues erroneous data, the security monitor can detect it by comparing the output against the healthy nodes. The erroneous node is

then isolated and the security monitor notes the compiler arguments that resulted in this defective machine code. The security monitor is not concerned with the root cause of the program error, but will attempt to correct the problem. Such a correction could be done by issuing different compiler arguments to permute the assumed faulty code, or rolling back to an earlier version that likely does not contain the faulty code.

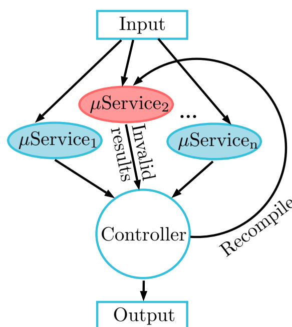


FIGURE 5.4: *A security monitor dealing with an infection in an N-version system.*

Referring back to the example with N-version programming, consider the case of removing an infection as indicated in Figure 5.4. The security monitor detects invalid data being sent from a service. The security monitor's presence on the host system is more privileged than the service itself. Hence, the security monitor is able to forcibly destroy the environment for the service, permute, and restore it. If the permutation step was skipped, the attacker could simply replay the exploit. The security monitor should proceed to flag the event as an anomaly to allow a human to examine the faulty binary to identify the underlying cause—which is likely only masked by the permutation.

SECURITY MONITOR POLICY. A simple policy for a security monitor service is to detect an intrusion, e.g. by using an IDS, kill the service environment, rebuild the environment, and finally restart the service. In this generalized procedure, the defender can either host the security monitor as a normal process with normal user privileges, in a container environment, or in a virtual machine. Regardless, the policy should be the same. It is important to destroy the whole environment, otherwise the risk of the attacker persisting increases dramatically. Even when destroying the environment the risk is only made smaller. If no containers are used, all processes should be removed and ideally the system (and firmware) restored from a trusted image—although even in this case advanced rootkits may persist. If containers or virtual machines are used, the entire container or virtual machine must be rebuilt. The permutation step ensures that diversity is added, which hopefully removes the issue.

The security monitor may choose to no longer trust the hosting machine for the

infected service, i.e. informing the assumed clean services to blacklist the malicious nodes as well as wipe and restore the system in an attempt to deal with a rootkit on the hosting machine. In addition, the security monitor can decide to destroy, permute, and restore all immediately adjacent services.

The security monitor system can be multi-layered. A local security monitor may reside in each execution context for each service, but preferably in a more privileged state so that a compromised service cannot trivially compromise the security monitor. However, an additional external security monitor is also possible. An external security monitor would enable more complex evaluations and actions being taken as a result of the state of the overall system, as compared to merely the state of a single node.

A HONEYPOT STRATEGY. Another possible strategy for the security monitor is to start a new node and ignore, but record the I/O of the infected node, as well as monitor it through the host system. The defender would be able to learn information about the attacker—in particular exploitation attempts—as the attacker is likely to continue to interact with the system. Such a honeypot strategy could be implemented to varying degrees of sophistication. All requests could be ignored, or some could be simulated, such that the attacker would continue to interact with the simulated environment, but not be able to gain any valuable asset or do damage. In the case of multiple infected nodes, a segment of the system could be isolated. Regardless, the defender should then also migrate away any other services running on the same infected host(s). There is always the risk that the attacker could escape the VM and take control over the whole system. If the defender does not own the system and risks exposing a cloud provider and other clients of the same provider to a known malicious attack, it seems at least plausible that this situation could lead to legal issues.

5.4.2 Evaluating the Security Monitor

In terms of the overall system architecture, the security monitor becomes a part of the infrastructure similarly to logging, monitoring, and discovery services that are needed for any reasonably sized microservice system to function properly. While the circuit breaker pattern aims to make systems more robust by preventing cascading failures, the security monitor pattern aims to make them more anti-fragile.

The security monitor scheme essentially allows the system to autonomously discover certain security related issues and react to them. Manual interaction is still required to resolve the root cause of the issue. However, at the same time the microservice architecture ensures that more effort is required to compromise the overall system, which makes the system more secure.

A more privileged mode that offers an attack surface is an ideal target. Indeed, the security monitor is such a target itself. IDS systems and anti-malware solutions have previously become a viable attack surface which raises the question whether such systems do more harm than good [91]. An IDS is always a trade-off, to prevent it from exposing the system to more risk rather than protecting it, the security

monitor should adhere to the aforementioned principles from Section 5.3.2 of least privilege, minimal attack surface, and have any grammar be of the highest type in the Chomsky hierarchy [113].

5.5 Proof of Concept by Example

This section presents a simplified banking system with minimal functionality and describes two attacks: one against the monolithic variant and a similar attack against the microservices variant of the system. We demonstrate that the microservice architecture makes a system more robust to the impact of attacks.

5.5.1 System Architecture

The system architecture contains four logical components: *Gateway*, *Users*, *Accounts*, and *Transactions*. *Gateway* provides the user interface and access to the functionality of the rest of the system. *Users* hosts the users' database and provides functionality for fetching user information and managing users. *Accounts* hosts the accounts database and provides account management operations. This service also has an IDS for demonstration purposes that reacts when a threshold is exceeded. *Transactions* hosts the transactions database and enables the creation of new transactions on demand. In the monolithic version of the system, all the components are contained within the same program.

The system is only presented in brief, as the details are not relevant to the issue being explained. The same concepts would apply for any similarly designed system, regardless of implementation details. The system is written in C and for simplicity uses raw sockets. The core functionality is a simple text-based user interface supporting basic transactions and account management. To illustrate the functionality, the following example shows a transaction being performed by a user after having logged in.

```
> view accounts
User ID: 1
Authorization: 0
Listing all accounts.
Account ID 1: Balance: 1000.000000
Account ID 2: Balance: 1000.000000
> pay 1 2 100
Transaction completed successfully.
> view transactions
Transaction ID 1, date: 2017-01-02 09:49:24: From account: 1 To
account: 2
Amount: 100.000000
>
```

It is assumed that the attacker has a copy of the source code of the program and knows the environment under which it was built and is presently executed. In

the following subsection, the first attack demonstrates how the attacker gains full control of the entire monolithic system. The second attack demonstrates that the attacker is only able to partly gain control of the microservice solution.

5.5.2 Exploitation Example

The goal of both attacks is read/write access to the accounts database. The attacker wants to manipulate the amount of money in a particular account. To achieve this goal, the attacker exploits a vulnerability in the server, obtains a shell, and finally interacts directly with the accounts database.

Attacking the monolith version. The attacker exploits the server using a stack based buffer overflow using a standard ROP (Return-oriented Programming) based exploit with the target having ASLR and NX-bit enabled. The exploit overflows a 512 byte buffer, hijacks the instruction pointer, and uses the stack to execute a set of gadgets (snippets of code contained in the target program which are carefully selected to execute a shell). Once the shell is obtained, the attacker spawns an interactive shell using Python to allow the sqlite3 utility to work. It can then be seen how the attacker leverages the fact that the asset in question (the accounts database) is readily available and can be directly manipulated with the privileges of the bank. The following is a session showing such an attack.

```
$ ./mono_exploit.py
[+] Opening connection to localhost on port 31337: Done
[*] Switching to interactive mode
Welcome to the Elite Bank

user: $ python -c 'import pty; pty.spawn("/bin/bash");'
<service_network/research/banking_system_monolith $ $ sqlite3
bank.db
sqlite3 bank.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> $ select * from accounts;
select * from accounts;
1|1000.0|2
2|1000.0|2
sqlite> $ update accounts set balance=9999999 where id=2;
update accounts set balance=9999999 where id=2;
sqlite> $ select * from accounts;
select * from accounts;
1|1000.0|2
2|9999999.0|2
sqlite> $ .exit
```

Attacking the microservice version. In the next example, the attacker uses the same exploit, only adjusted for the different gadget offsets for the microservice version.

```
$ ./microservice_exploit.py
[+] Opening connection to localhost on port 31337: Done
[*] Switching to interactive mode
Welcome to the Elite Bank.

user: $
$ nc -lp 2000 > steal_money.py
$ python steal_money.py 1
Success!
$ python steal_money.py 1
Success!
$ python steal_money.py 10
Success!
$ python steal_money.py 986
Failed!
$
```

It can be seen that the attacker again obtains a remote shell. However, the asset is not present on the server, therefore direct manipulation of the database is not possible. The attacker does, however, gain the ability to issue payment operations without proper authentication. Once having obtained the shell, the attacker immediately uploads a script to the compromised server which is used to steal money. The attacker specifies the amount of money to steal with a command line argument. This script interacts directly with the accounts service by connecting to it and issuing commands. This interaction would not be possible without having established a shell on the gateway node since the accounts service only trusts the gateway service. Connection attempts from the attacker would simply be dropped. However, the IDS employed by the accounts service detects the suspicious behavior and limits the damage.

Discussion. The attacker could at this point attempt to use a secondary exploit and move laterally within the microservice nodes, but this would require more effort from the attacker as compared with the monolithic version. The attacker could also try to avoid triggering the IDS. However, the defender has not lost the control flow of the whole system and has the possibility to mitigate such attacks.

There are obviously several ways to restrict access to the asset and achieve the same security benefits, depending on the application architecture. However, such mitigations would have to be tailored to the particular application. With a microservice based architecture, the security benefit is gained as a side effect of the architecture itself.

It should also be noted that the likelihood of the same bug existing in the microservice gateway node is less when directly compared with its monolithic counterpart. In addition, the code base for the microservice should be smaller, which could complicate the exploitation if key gadgets are missing. In this example, the required gadgets have simply been manually added to facilitate easy exploitation. Although, even without any of the critical gadgets, an arc injection or data-oriented exploit could still be performed in some cases.

5.6 Conclusion

We have examined how the increased isolation of microservices coupled with software diversity can mitigate the impact of low-level exploitation. Microservices, when coupled with some method of achieving diversification, appears to offer added robustness over monolithic solutions. Key design rules and examples were presented to substantiate this claim. Furthermore, an implementation of an example system demonstrated that a microservice solution is less vulnerable to low-level attacks than a deployment monolith.

We claim that the slow turnaround time for issues to be detected, fixed, and finally deployed by human operators can be made more autonomous and with lower latency if we introduce an automated security monitor to resolve the issues. One of the open questions that still remain is determining to what extent arbitrary programs can benefit from hardening and diversification. It is particularly important to consider the cost as most security enhancing features introduce overhead in terms of performance, compatibility, or usability, the mitigations suggested herein being no different.

References

[17] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. *N-Variant Systems A Secretless Framework for Security through Diversity*. 2006.

[19] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. “Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM”. In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS ’13. Hangzhou, China: ACM, 2013, pp. 299–310. DOI: 10.1145/2484313.2484351.

[26] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. “Microservices: yesterday, today, and tomorrow”. In: *CoRR* abs/1606.04036 (2016).

[32] C. Fetzer. “Building Critical Applications Using Microservices”. In: *IEEE Security Privacy* 14.6 (Nov. 2016), pp. 86–89. ISSN: 1540-7993. DOI: 10.1109/MSP.2016.129.

[37] M. Fowler. *Microservice Trade-Offs*. July 2015.
URL: http://martinfowler.com/articles/microservice-trade-offs.html (visited on Apr. 13, 2016).

[47] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson.
"ILR: Where'd My Gadgets Go?"
In: *Security and Privacy (SP), 2012 IEEE Symposium on*. May 2012, pp. 571–585.
DOI: 10.1109/SP.2012.39.

[54] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz.
"Compiler-Generated Software Diversity". In:
Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats.
Ed. by S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang.
New York, NY: Springer New York, 2011, pp. 77–98.
DOI: 10.1007/978-1-4614-0977-9_4.

[67] O. Lysne, K. J. Hole, C. Otterstad, Ø. Ytrehus, R. Aarseth, and J. Tellnes.
"Vendor Malware: Detection Limits and Mitigation".
In: *Computer* 49.8 (Aug. 2016), pp. 62–69. ISSN: 0018-9162.
DOI: 10.1109/MC.2016.227.

[80] F. Montesi and J. Weber.
"Circuit Breakers, Discovery, and API Gateways in Microservices".
In: *CoRR* abs/1609.05830v2 (2016). URL: http://arxiv.org/abs/1609.05830v2.

[85] S. Newman. *Building Microservices*. O'Reilly Media, 2015. ISBN: 9781491950357.

[86] NGINX, Inc. *The Future of Application Development and Delivery Is Now*.
Nov. 2015. URL: https://www.nginx.com/resources/library/app-dev-survey/
(visited on Jan. 6, 2017).

[89] M. Nygard. *Release It! Design and Deploy Production-ready Software*.
Pragmatic Bookshelf Series. 2007. ISBN: 9780978739218.

[91] T. Ormandy. *FireEye Exploitation: Project Zero's Vulnerability of the Beast*.
Dec. 2015. URL: https://googleprojectzero.blogspot.no/2015/12/fireeye-exploitation-project-zeros.html (visited on Feb. 7, 2017).

[96] D. Partridge and W. Krzanowski.
"Software diversity: practical statistics for its measurement and exploitation".
In: *Information and Software Technology* 39.10 (1997), pp. 707–717.
ISSN: 0950-5849. DOI: 10.1016/S0950-5849(97)00023-2.

[98] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis.
"Microservices in Practice, Part 1: Reality Check and Service Design".
In: *IEEE Software* 34.1 (Jan.–Feb. 2017), pp. 91–98. DOI: 10.1109/MS.2017.24.

[107] C. Richardson and F. Smith. *Microservices: From Design to Deployment*.
NGINX, Inc., 2016. URL:
https://www.nginx.com/resources/library/designing-deploying-microservices/.

[113] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina. "The Halting Problems of Network Stack Insecurity". In: *;login:* 36.6 (Dec. 2011).

[116] H. Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313.

[132] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 157–168. DOI: 10.1145/2382196.2382216.

[138] O. Zimmermann. "Microservices tenets: Agile approach to service development and deployment". In: *Computer Science - Research and Development* (2016), pp. 1–10. ISSN: 1865-2042. DOI: 10.1007/s00450-016-0337-0.

Paper III

A Game of Microservices: Automated Intrusion Response

Tetiana Yarygina, Christian Otterstad

6

© 2018 Springer.
Reprinted, with permission, from: T. Yarygina and C. Otterstad. "A Game of Microservices: Automated Intrusion Response". In: *Distributed Applications and Interoperable Systems*. Ed. by S. Bonomi and E. Rivière. Cham: Springer International Publishing, June 2018, pp. 169–177. DOI: 10.1007/978-3-319-93767-0_12

A Game of Microservices: Automated Intrusion Response

Tetiana Yarygina, Christian Otterstad

Department of Informatics, University of Bergen, Norway

Abstract The microservice architecture is a subtype of distributed systems that has been attracting a lot of interest both in the industry and academia. Emerging research recognizes the need for a better understanding of microservice security, and, in particular, mechanisms that enable defence-in-depth and adaptive security. With the continuously growing number of possible attacks and defenses, the choice in the optimal defense strategy becomes non-trivial as well as time critical. We propose a cost-sensitive adaptable intrusion response system for microservices, which uses a game theoretic approach to automatically respond to network attacks in real time. Furthermore, we discuss both the applicable attacks and defense responses specific to microservices.

Keywords *adaptive security, self-protection, game theory, defense-in-depth, SOA, IPS, IDS, minimax*

6.1 Introduction

Microservice architecture is gaining significant attention both by practitioners and in academia [85, 98]. Microservices allow for building flexible systems where the components can be written in different programming languages, use different technologies, scale independently, and can be easily updated and redeployed. Many microservice architectural principles such as modularity, loose coupling, and fail-fast are not new and stem from fundamentals of distributed systems [125]. Microservices are a particular implementation approach to Service-Oriented Architecture (SOA) [138]. However, the scale of microservice adoption is unprecedented and can perhaps be compared with the invention of object-oriented programming (OOP).

A key aspect of microservices is automation. With hundreds or thousands of microservices, manual updates are infeasible. Centralized logging, performance monitoring, service discovery are examples of such automation that are ubiquitously

adopted. Such trends as contentious integration, DevOps culture, and need for high scalability and flexibility further increase the importance of automation in microservice networks. Yet, not all the areas of microservice operation are automated.

Self-protection and other self-* properties, such as self-configuration and self-optimization that allow software systems to adapt to the changes in the environment, are actively researched areas [137]. However, the problem of self-protection for microservices has received scant attention in the research literature.

Regarding microservice security, Fetzer [32] discussed deployment of microservices inside secure containers to build critical systems. Sun et al. [123] proposed a virtual network monitoring solution that enforces policies over the network traffic in the cloud. Otterstad & Yarygina [93] pointed out the isolation benefits of microservices, as well as proposed the use of N-version programming for a simplified IDS. Yarygina & Bagge [135] have investigated automation of secure inter-service communication and defense-in-depth principle for microservices.

Intrusion detection systems (IDSs) provide system administrators with information on malicious activity in the system. While intrusion prevention systems (IPSs) attempt to block intrusions that are detected, the handling of complex situations and choice of intrusion responses are often left to humans. Once an intrusion is detected, actions should be taken as fast as possible to stop an attack.

According to Stakhanova et al. [121], an ideal intrusion response system (IRS) should be automatic, proactive, adaptable, and cost-sensitive. One of the possible ways of achieving such an IRS is through security games. Game theory [92] studies mathematical models of how multiple agents act when optimizing their payoffs. While the conventional game theory has a variety of applications in economics, political science, biology, it also received significant attention in the area of network security [88, 111, 139].

There is a general lack of research in IRS for microservices. With the continuously growing number of possible attacks and defenses, the choice of the best defense strategy is complicated. A game theoretic approach potentially solves this problem. Our main contribution in this paper is the design of a cost-sensitive automatic IRS for microservices with game-theoretic foundation; we also elaborate on the response actions specific to microservice architecture.

This paper is organized as follows. Section 6.2 explains the game theory fundamentals and defines the game model used in this paper. In Section 6.3, the architecture of the proposed IRS is presented. Section 6.4 evaluates the proposed architecture. Section 6.5 concludes the paper.

6.2 Security Games: Assumptions and Solutions

We observe that the microservice architecture readily allows for employing game theory derived algorithms. This is a foundation for the system ability to respond to intrusions. This paper models the strategic interaction between an attacker and a defender as a finite dynamic zero-sum game. Different game theoretic solution concepts exist [92].

6.2.1 Finite Dynamic Two-player Zero-sum Game

Most security games involve two counteracting parties—an attacker and a defender. While attacker's goal is to exploit the system, a defender is trying to protect the system, its resources, and data. Each player has a set of actions available to them at given time. Each action has a positive or negative reward associated with it. In the case of attacker and defender, each player's gain or loss is balanced by the losses or gains of the other player, which makes this game *zero-sum*. A compromised microservice node has a negative score for the defender, but a positive score for the attacker.

The attacker and defender take actions and receive rewards in turns, as seen in Figure 6.1. In this way, the game moves from one security state to the other. Games with more than one stage are called *dynamic* or *extensive*.

There is a limited number of states a given system can be in. Some of the states are the final states, i.e. the leaf nodes on the game graph. The security states where the attacker gains full control of the system can be seen as final states. In such case, the game is called *finite*.

6.2.2 Minimax

The goal for the defender is to choose the optimal response action in the given context. A common solution to this problem is the minimax algorithm. The minimax strategy for a defender is a strategy that minimizes the maximum payoff of an attacker, i.e. maximizes the benefit of a defender. For two-player finite zero-sum games, the minimax solution is the same as the Nash equilibrium.

In a two-player game, the minimax strategy for a defender i against an attacker $-i$ is $\arg\min_{a_i \in A_i} \max_{a_{-i} \in A_{-i}} u_d(a_i, a_{-i})$, where a_i is the response action taken by defender, a_{-i} denotes the attack action taken by adversary, and u_d is the utility function of defender.

Completely analyzing certain games using solely the minimax algorithm is impractical. The minimax algorithm traverses the nodes of a game tree. The number of nodes on the tree increases exponentially with the number of turns that can lead

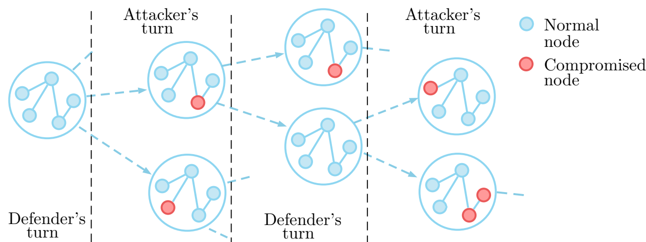


Figure 6.1: *Security game between the defender and attacker. An attacker decides which microservice to attack. The response actions deployed by the defender may or may not eradicate the attacker. If defense measures are insufficient and/or unsuccessful, the attacker will propagate laterally through the network.*

to a combinatorial explosion. The performance of the minimax algorithm can be improved using alpha-beta pruning or other pruning methods.

6.3 Proposed Architecture

Building on this game theoretic foundation, we propose a microservice intrusion response system. The system is cost-sensitive, in that the game allows us to chose the most effective, least costly response to an attack, rather than applying drastic measures in every situation.

The system consists of a distributed set of local monitoring processes and a central entity called the Microservice Game Engine (μGE), as depicted in Figure 6.2. The purpose of the system is to minimize the damage caused by an attack in real time. The μGE allows the microservice network to dynamically react to threats while taking action costs into consideration. In particular, the μGE exploits the fact that a partially compromised microservice network has not yielded total control to the attacker. A strong separation of the control flow and increased isolation are inherent benefits of microservice architectures [93, 135].

The μGE aggregates information, builds a game tree and takes automated action based on the observed input obtained from the local IDSes running on the respective microservices. In order to facilitate these actions, the μGE relies on several key components discussed below.

6.3.1 Network Mapping

The ability to maintain an accurate view of the microservice network at all times is a prerequisite for the μGE to respond effectively to malicious activity. The initial microservice network can be mapped in several ways. For example, each microservice can report its incoming and outgoing edges to the μGE.

After the network has been initially built by the μGE, it is ready to start playing the game in anticipation of attacks. However, when the real network changes, its representation inside the μGE should also be updated, and all computation performed thus far will be discarded. Nodes are inserted and removed from the tree as they report to the μGE.

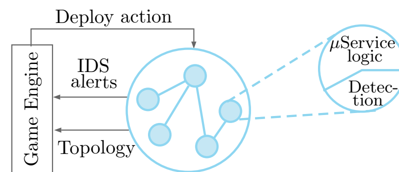


FIGURE 6.2: *An overview of the proposed architecture. The detection code in each microservice reports to the game engine, which evaluates the current state of the system and plans a response to any ongoing attack. Responses are deployed by e.g., restarting or reinstalling services, or other response actions.*

6.3.2 Intrusion Detection and Events Reporting

Intrusion detection functionality resides in each microservice, informing the μGE of events of interest. Multiple sources of IDS information, as well as non-IDS information may be used. An IDS such as Snort [120] can detect among other things stealth port scans, operating system fingerprinting, and buffer overflows. Non-IDS information include events in the network, such as a service having stopped sending heartbeats, service registration and deregistration.

Attack actions in the game can be defined through the intrusion detection function. The set of possible attack actions is limited to the knowledge base of the particular IDS in place.

6.3.3 Event Evaluation Function

The evaluation function takes as its argument a node object representing a state of the microservice network. The state of this network in terms of score is then evaluated and returned as an integer. The most basic and coarse node states may be grouped into three categories. *Benign:* default normal state of all nodes. *Under attack:* an alert for the node has been raised. *Compromised:* a node that conducts malicious activity or is suspected of one.

The μGE can notice that events have happened over time, and/or that events are happening in multiple places of the network at the same time—each event which by itself is not enough to trigger an issue, but as an aggregated result. The aggregated result may be accumulated in a temporal and/or spatial sense. This is similar to what distributed IDSs would do, see the survey by Folino [36].

The evaluation function should be able to aggregate information such that a node can be inferred to be compromised based on its behavior. E.g. if there is a port scan or API probing attempt from an internal node, this is assumed to only be possible if the node is compromised. The node may therefore be flagged as compromised even though no direct detection of an attacker present on the system was detected. If there is only one other node which could communicate with the compromised node, the evaluation function can further infer that this node is also compromised. By extension, the μGE can infer a chain of guaranteed compromised nodes and possibly compromised nodes. As an example, if there are two additional nodes which can communicate with the node, the evaluation function can trivially assume with 50% probability that either of them are compromised, assuming everything else is equal.

6.3.4 Decision Function

The decision function runs a minimax algorithm with specific pruning mechanisms. If there is no actual malicious action and no network related update taking place, the μGE will populate the tree representation of different possible states. For each particular node of the microservice network, based on the state of the network, there will be a list of possible actions the attacker or defender can perform. The set of possible actions is used to create new states of the same microservice network, creating

new nodes, where the edges from the previous node represent the particular action taken.

Whenever any new information which results in a different state is received, the μGE stops its search, and updates the root of the tree to be that resulting state. This simulates performing the actual operation the attacker did against the real network. It is now the defender's turn. The μGE will also run the evaluation function on all the leaf nodes and compute the optimal move for the defender. We make the conjecture that the time taken to actually run this part of the algorithm is negligible and can be run post attack. The best strategy is chosen based on the available response actions discussed below.

6.3.5 Intrusion Response: Defender's Actions

Traditional fault tolerance techniques include rollback, rollforward, isolation, reconfiguration, and reinitialization [5]. Microservice based systems, however, allow more actions to be taken. In the case of an assumed compromised node, the defender may opt for the following choices.

Rollback/restart the service. This will destroy the current instance of the service and start a new one from the same configuration. If the problem persists, even older configuration of the service may be used. This allows the defender to hopefully mitigate attacks based on flawed configuration assuming it was ever correct, as well as bugs introduced in the latest version.

Diversification through recompilation or binary rewriting. Introducing randomness into the binaries executing in the microservice may be done by binary rewriting or recompilation with special compiler support [54]. An example of a freely available framework providing such support is the LLVM multicompiler.

Diversification through cloud provider. Moving a microservice to a different host in attempts to mitigate attacks that rely on host characteristics, such as exploits that target hardware or a malicious cloud provider.

Scale up and down n-variant services. The N-variant microservice system was proposed as a security measure by Otterstad&Yarygina [93]. This action uses the existing diversification techniques (compiler diversity/binary rewriting and cloud diversity) to spawn additional microservices, which feed their result to a governor node that compare the results for consistency. This allows nodes that have been tampered with to be detected.

Split or merge services. Requires a tool support for code auto modification that does not currently exist. An extension of this approach is to add dummy hardened services to the path. A node may be split at the function level, this may mitigate certain attacks that rely on there being a binary path of execution which enables the exploit to work, and/or the existence of certain gadgets, which will not be available after a split has been performed.

Isolation/Shutdown. Entails physical exclusion of the faulty service: stop the service permanently. This approach has a high cost associated with it and is unacceptable for the systems with high availability requirements. Noureddine et.al. [88]

showed that disconnecting nodes in a response to an attack efficiently delays an attacker that is moving laterally in the simulated network.

6.4 Evaluation and Discussion

Advantages. The controller system has two main advantages: low latency, and depth. In contrast to a human, who may need time to understand and react to the attack, the μGE can react instantly. The latency is important due to the fact that some attacks may be automated and complete a sequence of steps in the attack very quickly, which would make it critical to be able to respond promptly.

The depth is important for related reasons. The μGE can search deeper into the tree and gain a deeper insight than a casual observation from a human. Some choices may not seem intuitive as they result in a better network state deeper down in the tree. It may be possible that the attacker performs a particular attack to which the naive reaction is what allows the actual attack to proceed.

Algorithm complexity. The depth of the tree is m with b legal moves on average (the branching factor of the tree). The running time of the minimax algorithm is $O(b^m)$, and space complexity (memory) is $O(b * m)$. For large trees, high complexity can make the approach computationally infeasible. For the alpha-beta pruning algorithm, sorting the moves by the result improves performance, such that for the perfect ordering the running time is $O(b^{m/2})$.

Model limitations. So far, we discussed only a subset of all possible attack and defense actions. In a real world scenario there are not only more nodes, but many more operations the attacker and defender could do. This causes an explosion in the complexity of the tree, which greatly limits the depth of the search, consumes much more memory, and CPU time.

The list of attacker and defender operations is a model of the real world. Any such model will have limitations in terms of granularity. The extent to which the defender is willing to exert resources on creating an accurate representation of the real world will affect the effectiveness of the system. However, even for missing classes of attacks, they are likely to result in a state which the system will detect. Lets consider a node compromised with a zero-day exploit that went unnoticed. It is extremely unlikely the attacker has a zero-day for every node in the network. Thus, when the attacker starts to probe the rest of the network from an internal node, looking for well-known vulnerabilities, the μGE will again notice the issue and can consult its graph for the optimal course of action.

This paper assumes that the utility function and reward values are designed by experts offline. Selecting the best defense model is difficult because of a lack of quantifiable security metrics. Despite the multiple attempts to address this problem [118], putting values to parameters is still a human responsibility.

6.5 Conclusions

This paper presented the design of a cost-sensitive adaptable IRS applicable to microservice networks called μGE. The μGE collects information from the network as well as issues actions based on a search tree of possible outcomes once an attack has been detected. The proposed solution exploits the fact that the microservice network is modular by design and components can be restarted, permuted, moved, and even in some cases removed, without destroying the entire operation of the network. In general, no mitigation technique provides a guarantee an attacker cannot succeed. However, the μGE enables low latency and far lookahead which is a strong advantage for a defender.

Several open questions remain. An efficient approach to identifying and setting the values that are topical to each defender has not been presented, as this is highly subjective and specific to the assets that are important. Furthermore, for an algorithm of this type to be efficient on big networks, it is likely that a significant amount of aggressive pruning of the search tree must be performed.

References

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[32] C. Fetzer. "Building Critical Applications Using Microservices". In: *IEEE Security Privacy* 14.6 (Nov. 2016), pp. 86–89. ISSN: 1540-7993. DOI: 10.1109/MSP.2016.129.

[36] G. Folino and P. Sabatino. "Ensemble Based Collaborative and Distributed Intrusion Detection Systems". In: *J. Netw. Comput. Appl.* 66.C (May 2016), pp. 1–16. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2016.03.011.

[54] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. "Compiler-Generated Software Diversity". In: *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats.* Ed. by S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang. New York, NY: Springer New York, 2011, pp. 77–98. DOI: 10.1007/978-1-4614-0977-9_4.

[85] S. Newman. *Building Microservices.* O'Reilly Media, 2015. ISBN: 9781491950357.

[88] M. A. Noureddine, A. Fawaz, W. H. Sanders, and T. Başar. "A Game-Theoretic Approach to Respond to Attacker Lateral Movement". In: *Proceedings of the 7th International Conference on Decision and Game Theory for Security (GameSec 2016).* Ed. by Q. Zhu, T. Alpcan, E. Panaousis, M. Tambe, and W. Casey. Springer International Publishing, 2016, pp. 294–313. DOI: 10.1007/978-3-319-47413-7_17.

[92] M. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994. ISBN: 9780262650403.

[93] C. Otterstad and T. Yarygina. "Low-Level Exploitation Mitigation by Diverse Microservices". In: *Service-Oriented and Cloud Computing (ESOCC 2017)*. Ed. by F. De Paoli, S. Schulte, and E. Broch Johnsen. Oslo, Norway: Springer, 2017, pp. 49–56. DOI: 10.1007/978-3-319-67262-5_4.

[98] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. "Microservices in Practice, Part 1: Reality Check and Service Design". In: *IEEE Software* 34.1 (Jan.–Feb. 2017), pp. 91–98. DOI: 10.1109/MS.2017.24.

[111] S. Roy, C. Ellis, S. Shiva, D. Dasgupta, V. Shandilya, and Q. Wu. "A survey of game theory as applied to network security". In: *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*. IEEE. 2010, pp. 1–10.

[118] C. B. Simmons, S. G. Shiva, H. S. Bedi, and V. Shandilya. "ADAPT: a game inspired attack-defense and performance metric taxonomy". In: *IFIP International Information Security Conference*. Springer. 2013, pp. 344–365.

[120] *Snort official web-site*. URL: https://www.snort.org (visited on Feb. 23, 2018).

[121] N. Stakhanova, S. Basu, and J. Wong. "A taxonomy of intrusion response systems". In: *International Journal of Information and Computer Security* 1.1-2 (2007), pp. 169–184.

[123] Y. Sun, S. Nanda, and T. Jaeger. "Security-as-a-service for microservices-based cloud applications". In: *Cloud Computing Technology and Science (CloudCom 2015)*. IEEE. 2015, pp. 50–57.

[125] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007. ISBN: 9780132392273.

[135] T. Yarygina and A. H. Bagge. "Overcoming Security Challenges in Microservice Architectures". In: *12th IEEE International Symposium on Service-Oriented System Engineering (SOSE'18)*. Bamberg, Germany: IEEE, Mar. 2018, pp. 11–20. DOI: 10.1109/SOSE.2018.00011.

[137] E. Yuan, N. Esfahani, and S. Malek. "A systematic survey of self-protecting software systems". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 8.4 (2014), p. 17.

[138] O. Zimmermann. "Microservices tenets: Agile approach to service development and deployment". In: *Computer Science - Research and Development* (2016), pp. 1–10. ISSN: 1865-2042. DOI: 10.1007/s00450-016-0337-0.

[139] S. A. Zonouz, H. Khurana, W. H. Sanders, and T. M. Yardley. “RRE: A game-theoretic intrusion response and recovery engine”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.2 (2014), pp. 395–406.

Paper IV

RESTful Is Not Secure

Tetiana Yarygina

© 2017 Springer.
Reprinted, with permission, from Tetiana Yarygina, *RESTful Is Not Secure*, In: Batten L., Kim D., Zhang X., Li G. (eds) 8th International Conference on Applications and Techniques in Information Security. ATIS 2017. Communications in Computer and Information Science, vol 719, pp.141-153. Springer, Singapore. DOI: 10.1007/978-981-10-5421-1_12..

RESTful Is Not Secure

Tetiana Yarygina

Department of Informatics, University of Bergen, Norway

Abstract The shift in web service design towards the REST paradigm has spawned a series of security concerns. To date there has been no general agreement on how the REST paradigm addresses security and what web security mechanisms adhere to the REST style. This paper analyzes the REST paradigm from a security perspective and shows significant incompatibilities between the style constraints and typical security mechanisms. We conclude that the REST style was not designed with security properties in mind and does not fit the security requirements of modern web applications.

Keywords *web services security, REST, stateless, token authentication*

7.1 Introduction

Web services enable rapid design, development, and deployment of software solutions. They provide a unified web interface and hide complexity and heterogeneity of the underlying infrastructure, enabling simple integration of diverse clients and external components [108]. Unfortunately, the desirable simplicity does not extend to the security aspects of web services.

Representational State Transfer (REST) is an architectural style for web services that is widely adopted. As an architectural style, REST imposes six general design constraints [35]: *client-server*, *stateless resource*, *cacheable responses*, *uniform interface*, *layered*, and *code-on-demand* (optional constraint). These constraints enforce the original concept of the Web as a scalable distributed hypermedia system with loosely coupled components. Web services that strictly adhere to REST style constraints are commonly referred to as RESTful services, while those with loose adherence are often called REST-like services.

It was long believed [100] that RESTful services should be used for ad hoc integration over the Web, whereas Big Web services (see [108] for naming convention) were preferable in enterprise application integration scenarios with longer lifespans and advanced security requirements. However, today we find that more and more

corporate solutions, even the most security demanding ones like financial systems and sensitive data operations, are based on RESTful or REST-like services. In contrast to Big Web services, no formal security framework exists for RESTful services.

There are relatively few studies of RESTful services security, herein we mention most of them. A recent study by Gorski et al. [40] compares the security stacks of Big Web services and RESTful services. A paper by Lo Iacono and Nguyen [66] compares RESTful authentication mechanisms with focus on message signing. In particular, the authors propose a signing mechanism not limited to HTTP. Finally, two papers describe approaches to message security for RESTful services [115] and secure communication between mobile clients and RESTful services [21]. Although all of these studies claim to deal with RESTful security, they do not discuss the REST architectural style from security perspective.

A much debated question among practitioners is what security mechanisms are truly RESTful. As an example, discussion threads on RESTful authentication[1] and best practices for securing REST APIs[2] are viewed more than 250,000 times each. The introduction of security components often changes system behavior, which can affect how a system adheres to the REST style constraints. To date there has been no general agreement on how the REST paradigm should address security. Apart from Inoue et. al. [53], who argued that a session state is not against the REST architectural style, there is a lack of research in the area.

This paper aims to unravel some of the mysteries surrounding RESTful security. We analyze the REST paradigm from a security perspective and show significant incompatibilities between the style constraints and typical security mechanisms. To our knowledge, we are the first to conduct such a detailed security evaluation of the REST style and prove that RESTful security is impossible.

The rest of the paper is organized as follows. In Section 7.2, an overview of common web security mechanisms and a brief discussion of their security merits are given. Section 7.3 explores in detail how particular security decisions and especially authentication schemes relate to core principles of the REST style. Section 7.4 concludes the paper by summarizing the uncovered contradictions, discussing the implications of the findings, and providing insights for future research.

7.2 Overview of Security Mechanisms for the Modern Web

Adequate security mechanisms are needed to build secure RESTful services. This section focuses on common security mechanisms such as Transport Layer Security (TLS), cryptographic objects in JavaScript Object Notation (JSON), token-based authentication, client side request signing, and delegated authorization and shared authentication. The overview creates a background for a more advanced analysis of how common security mechanisms adhere to the REST style constraints.

TLS was originally designed to be independent of any application protocol and

[1] https://stackoverflow.com/questions/319530/restful-authentication

[2] https://stackoverflow.com/questions/7551/best-practices-for-securing-a-rest-api-web-service

has became a de facto security protocol on the Web. Although the design of TLS supports mutual authentication, HTTPS in its current form is largely used to authenticate the gateway, but not the client. Even though the idea of both parties maintaining digital certificates is simple and secure, embedding a unique certificate into each client is a serious implementation obstacle. Therefore, client authentication must be provided on the application (message) level.

To provide higher security, as well as client authentication, TLS can be and often is combined with encryption and signing on the message level. Standards for cryptographic objects in JSON and XML were created to address security needs on the message level and to facilitate interoperability. Cryptographic objects can be seen as containers incorporating secured data and the information necessary for its processing. The JSON Object Signing and Encryption (JOSE) suite of specifications offers powerful and flexible building blocks for message security in web services by providing a general approach to signing and encryption of JSON-formatted messages. The JOSE suite is essential for delegated authorization and shared authentication schemes, such as OAuth 2.0 and OpenID Connect (see Figure 7.1).

HTTP is a stateless protocol, which implies that requests are treated independently of each other. Nevertheless, most web applications require sessions. Session management in HTTP is historically performed via HTTP cookies, URL parameters, HTTP body arguments in requests, or custom HTTP headers. A natural extension of session management is client authentication. In modern web applications, there exist two main approaches to authentication: *token-based authentication* and *client side request signing*. The following discussion focuses on the security aspects of these approaches.

Traditionally [74], message authentication methods include Message Authentication Codes (MACs), digital signature schemes, and appending a secret authenticator value before encrypting the whole text. In the context of modern web services, either JWS or XML Signature standards can be used for message authentication depending on the message format. For the sake of simplicity, the term signature is used to refer to both MACs and actual digital signatures.

7.2.1 Token-based Authentication

Token-based authentication via HTTP cookies is the most widely adopted authentication mechanism in web applications. The mechanism is based on a notion of

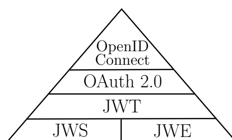


FIGURE 7.1: *The hierarchical relation between the JOSE suite, OAuth 2.0, and OpenID Connect. The JOSE suite incorporates JSON Web Signature (JWS), JSON Web Encryption (JWE), JSON Web Token (JWT) [55], and several other specifications.*

security tokens—cryptographic objects containing information relevant for authentication or authorization.

An authentication token is generated by a web service and sent to a client for future use. A service generates a token upon the successful validation of the client's credentials either during the initial user log in or a re-authentication. A token can be seen as a temporary replacement for the client's credentials: every request from a client must include a valid token to be fulfilled. A token-based authentication scheme was first analyzed by Fu et. al. in 2001 [39].

Security considerations. Server-created security tokens ensure scalability of the solution and server statelessness by moving the maintenance responsibility for tokens to clients. Additionally, a limited lifetime of security tokens makes them superior to direct use of passwords such as in HTTP Basic/Digest Authentication. A server-side secret used to create tokens is the most important security asset of the server. If the secret is leaked, the damage is not limited to one user: an adversary can impersonate any user of his or her choice.

Hijacking of security tokens is another serious threat. Token-based mechanisms rely on channel confidentiality. If compromised, a security token can be used by an adversary to impersonate the client until the token expires or is revoked. Short expiration time of tokens limits the possible damage, but also reduces usability of a system by requiring frequent user re-authentication.

The severity of security token hijacking is rooted in the static nature of such tokens and their independence of particular requests. Dacosta et. al. [18] proposed to switch from static cookies to dynamic ones (request-specific). Channel-binding cookies is another approach to strengthen cookie-based authentication by binding cookies to TLS channels using TLS origin-bound certificates [22]. However, no approach has gained wide adoption mostly due to increased complexity. The evidence presented herein suggests that token-based authentication requires minimal amount of data being stored on the server-side, i.e. contributes to server statelessness, but also has significant security limitations.

7.2.2 Client Side Request Signing

Many existing RESTful services implement client authentication and in-transit tampering protection by requiring a client to sign each request. Cryptographic keys are established between parties during or after the initial authentication step. Request signing implies signing of an actual message (HTTP payload) and, optionally, HTTP headers.

Request signing involving HTTP headers has been successfully deployed by several major web services such as Amazon Web Services (AWS) [2] and Microsoft Azure [75]. Both are cloud services intended only for programmatic use through REST APIs. An investigation shows that numerous newly developed systems borrow AWS' HMAC-SHA256-based approach to request signing [2].

A comparison of REST message authentication mechanisms based on request signing was performed by Lo Iacono and Nguyen [66]. The paper contributes a

detailed HMAC-based scheme for authentication of all types of REST messages, including HTTP messages. A similar, but not as detailed approach to HTTP signing can be found in the IETF draft *Signing HTTP Messages* [12].

Security considerations. Client-signed requests provide stronger authentication than mere token-based schemes. Signing of each client request effectively mitigates session hijacking attacks by limiting damage only to a single request. A signing key never leaves a client which makes stealing the key much more difficult than stealing a token that is not only stored on the client, but also repeatedly sent over the channel. As often happens, higher security comes at a price of lower scalability and higher complexity since a server needs to maintain a separate key for each user.

7.2.3 Delegated Authorization and Shared Authentication

Delegated authorization and shared authentication have become an integral part of modern web security. The popular security protocols underlying delegated authorization and shared authentication mostly instantiate the token-based authentication introduced earlier. Therefore, they share both advantages and disadvantages of token-based authentication.

Delegated authorization. We consider a scenario where a user, or resource owner, has stored some sensitive information on a server. The desire to separate the login process on the server from the process of granting permissions to a client application on the behalf of the user has stimulated the emergence of OAuth [44]. OAuth is a delegated authorization protocol providing third-party applications (clients) with delegated access to protected resources on behalf of a user (resource owner). Client side request signing in OAuth 1.0 enables client authentication and message integrity, while OAuth 2.0 does not. Developers often fail to implement OAuth correctly due to its ambiguity and complexity [13, 122, 131].

Shared authentication. OAuth 2.0 is used as an underlying layer for shared authentication protocols and Single-Sign-On (SSO) systems. Prominent examples are OpenID Connect [112], Facebook Login, and Sign In With Twitter. In such schemes the user authenticates into a third party service (a Relying Party or RP) using a digital identity at an Identity Provider (IdP) of the user's choice. However, additional steps must be taken in order to use OAuth 2.0 for authentication. Security analyses of commercially deployed OAuth-based SSO solutions (i.e. popular social login providers) [122, 130] have revealed various security and privacy issues.

7.3 REST Architectural Style and Security

So far this paper has focused on the security mechanisms commonly used to secure RESTful services. This section elaborates on why none of the systems using such mechanisms are strictly RESTful by analyzing the REST style and its constrains from a security perspective. It is worth mentioning that the majority of RESTful services actually fail to adhere to REST for reasons unrelated to security. Absence of custom media types support and use of verbs in URIs are common examples of such violations.

The REST architectural style was introduced by Fielding in his influential dissertation [35] and related paper [33] in 2000. The style is widely adopted and many popular web services, such as Twitter[3] and LinkedIn[4], have REST APIs. The dissertation remains the most fundamental source when talking about the core principles of REST.

7.3.1 Not Designed with Security in Mind

The REST style was proposed as an architectural standard for the Web and introduced only the properties that seemed necessary for the Web at that time. Fielding makes no attempt to address the question of security in REST. The words security, authentication, and authorization are rarely mentioned in Fielding's work. The words encryption and signing do not appear at all.

According to Fielding [35], "REST emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, *enforce security*, and encapsulate legacy systems." The claim that REST enforces security is neither justified in the dissertation nor explained in any other literature related to REST.

When talking about scalability of the Web, Fielding writes [35, Sec. 4.1.4.1] "since authentication degrades scalability, the architecture's default operation should be limited to actions that do not need trusted data." In modern Web, and especially for REST APIs, the situation is reversed: some form of authentication is always present. TLS is only mentioned as a connector type [35, Sec. 5.2.2], no encryption on the message level is considered.

The REST architectural style does not incorporate security as one of its goals and leaves it up to the developer to decide how security fits the six core principles. The introduction of security components affects system behavior initially shaped by REST constraints. Most of the constraints, such as client-server, uniform interface, and layered system, are high-level and flexible enough to not interfere with adopted security mechanisms. At the same time, the stateless, cacheable, and code-on-demand constraints have several practical security implications. The security implications of the relevant REST constraints are discussed in the following sections.

7.3.2 Stateless Constraint

Revisiting the definition. The stateless resource constraint is particularly problematic from a security perspective. The constraint is often misunderstood by practitioners and overlooked in the scientific literature. According to Fielding [35, Sec. 5.1.3], for a resource to be stateless "each request from client to server *must* contain all of the information necessary to understand the request, and cannot take advantage of *any stored context on the server*." Such a definition makes no exceptions and, when followed to the letter, leaves no room for security mechanisms.

[3] https://dev.twitter.com/rest/public

[4] https://developer.linkedin.com/docs/rest-api

Furthermore, [35] specifies that the "session state" (also referred to as "application state") should be stored exclusively on the client side; however, a definition of session state is never given. A commonly used interpretation of the stateless resource constraint introduced in [108] differentiates between *application state* and *resource state*. A resource state is defined as any information about the underlying resource [108].

While the resource state belongs to the server, it still can be changed in response to a client request. If we consider a user as a resource, then the balance of the user's bank account is a resource state that is changing with each performed transaction. Similarly, usernames and passwords are also resource states that change over time.

Security implications. Most security components introduce additional resource states. Stateless security protocols do not exist. It is very hard, if at all possible, to prevent replay attacks without maintaining at least some form of client state on the server side. Nonces (numbers used once), counters, and timestamps are examples of such a resource state. All authentication mechanisms described in Section 7.2 incorporate one or more such components. Thus, web services utilizing these mechanisms are not strictly RESTful.

Differentiating between application state and resource state can be difficult. For example, security tokens are stored by the client, but are issued exclusively by the server. The server must maintain the key(s) used to sign tokens, which introduces more resource states.

The demand of "taking no advantage of any stored context on the server" is impractical. For example, a common security practice of restricting the number of login attempts made per specific account relies on the login history being available.

As pointed out by Fielding [35, Sec. 6.3.4.2], HTTP cookies fail to fulfill the stateless constraint of REST. An example of such a violation is the use of cookies to identify a user's "shopping basket" stored on the server, while the basket can be stored on the client side and presented to the server only when the user checks out. This mismatch between REST and HTTP makes a huge part of the modern Web not RESTful and implicitly deprecates cookie-based authentication for RESTful web services.

When token-based mechanisms, such as JWT, OAuth 2.0, and OpenID Connect are used, a server needs $\mathcal{O}(1)$ resource states to authenticate N users [39]. With client request signing as in OAuth 1.0a and AWS, the server needs to maintain a separate key for each client, thus having $\mathcal{O}(N)$ resource states. Therefore, token-based mechanisms can be considered stateless in a sense that there is no per-user or per-session state when compared to client request signing given a substantial number of clients. Although token-based authentication fits the REST style better then the client side request signing, the latter is generally more secure as explained in Section 7.2.

Additionally, it is possible to classify application state into two classes, security insensitive and security sensitive, that must be treated differently. The server cannot prevent the client from tampering with the data given to it, nor can the server

directly protect data stored on a client from malicious third parties. The latter puts user privacy at risk if the data stored is security sensitive.

Even though the definition of the stateless constraint dictates that a client's request must contain all of the information necessary to understand the request, sensitive information should not be transferred unless absolutely necessary. All security sensitive application states must belong to the server and be resource states.

Advantages and disadvantages. To evaluate immediate importance of stateless resource constraint for modern security-aware applications, the advantages and disadvantages of the constraint must be revisited. According to Fielding [35], stateless resource constraint induces the properties of visibility, reliability, and scalability.

The original argument for improved visibility [35] was that the server should process a client request without looking beyond this request. The argument is valid until security is involved. Let us consider an online store. If some items are added to the shopping basket, the only allowed step should be a payment step, and not goods delivery. To ensure this restriction, the user must have state within the system.

Additionally, intrusion detection systems (IDS), anti-denial-of-service, and anomaly detection mechanisms are more likely to mitigate attacks when they have knowledge of the state and the history of requests. If we consider security sensitive data such as authentication tokens, the server unavoidably needs to validate the token, which requires retrieval of the cryptographic key used to generate the token. The step of token verification can also be seen as one that decreases visibility. The aforementioned suggests that improvement of visibility can only be seen for security insensitive data.

The common belief is that maintaining client states on the server side can potentially create a high load of session management and degrade system performance. However, storing clients states on the server side does not cause significant performance problems for existing high load systems and Cloud services; a study of REST session state [53] showed that the impact of the stateless resource constraint on scalability and reliability of REST in the modern Web is insignificant.

Moreover, maintaining client states on the server side is a desired property in many cases, for example personalized services, targeted advertisement, smart suggestion systems, and IDS benefit from it. An alternative solution to scalability and reliability issues is adoption of special software architecture styles, such as microservices [32].

The stateless constraint puts significant limitations on handling session synchronization. In the example with the shopping basket, the problems occur when the user has initialized a session on a mobile device and wants to continue the session using the browser on a laptop. Storing session state exclusively on the client side and not on the server makes it impossible to keep persistent state in such situations. Hence, current demand for client state synchronization negates the stateless resource constraint of REST.

7.3.3 Other Constraints Affecting Security

Cache constraint. The cachebility constraint is affecting security much less than the stateless criteria, but the effect is still noteworthy. The definition of the constraint [35] states that the server responses must be explicitly marked as cacheable or noncacheable. Of course, only actual caching of responses improves scalability and network efficiency by eliminating identical repeating interactions. Caching of server responses can be performed by intermediates, i.e. proxies and gateways, or clients themselves.

Caching by intermediates has less value on the modern Web due to an increasing amount of encrypted traffic such as HTTPS traffic. As of February 2017, 52.8% of the most popular websites implemented HTTPS [127]. When encrypted either by TLS or on the message level, server responses are not cacheable by intermediate proxies. Encrypted content cannot be cached unless the intermediates are allowed to decrypt the traffic, which defeats the purpose of encryption in the first place.

Although caching by clients is not affected by encryption, it loses its importance due to different reasons. Modern websites include large amounts of dynamic personalized content that cannot and should not be cached. In case of online banking or online shopping the content (the bank account balance or availability of specific items in the shopping basket) is dynamic and gets outdated fast. Such content is not suitable for caching due to reliability reasons. Similarly, sensitive content should never be cached for security reasons.

Taken together, encryption and personalized content dramatically reduce the benefits of traditional web caching in general, and the importance of cache constraint of the REST style in particular. While the content marked as noncacheable does not contradict the definition of the cache constraint (since the constraint only requires proper labeling), it brings no actual benefit in terms of scalability or network efficiency.

Code-on-demand constraint. In the code-on-demand paradigm the code for a specific task is requested by the client, provided by a server, and executed in the client's context. As argued in [35], the code-on-demand constraint of REST improves system extensibility, but also reduces visibility. Therefore, it is only an optional constraint.

It should be noted that the code-on-demand constraint is relevant primarily within the browser environment. In semantic web with machine-to-machine communication and native clients consuming REST APIs, execution of external JavaScript code in the native applications is currently uncommon.

An important security implication of the code-on-demand paradigm is an increased attack surface on a client. Among the major security concerns are authenticity of the received code and the client's ability to limit the behavior of the code. These problems have been studied for a long time and mitigation techniques, such as sandboxing, Address Space Layout Randomisation (ASLR), and Data Execution Prevention (DEP), are implemented in modern browsers. However, the problems

still persist.

7.4 The Way Forward

7.4.1 Security Failure of REST

The main goal of this paper was to asses how the REST style addresses security and whether security mechanisms adhere to the style constraints. The study has shown that the REST style fails to take security into account, or to explain security implications of the constraints. To fill the gap, we provided the missing security interpretation of the relevant style constraints and made the following observations:

- *Stateless resource constraint.* The more security critical a system is, the more resource states it is likely to have. Among authentication approaches, token-based authentication most closely fits the stateless resource constraint. However, it is not entirely stateless.

- *Cache constraint.* Although formally the cache constraint (labeling of responses) is not directly affected by security mechanisms, the constraint loses its meaning for security critical systems. Encrypted, dynamic, and personalized content is not suitable for caching.

- *Code-on-demand constraint.* The optional code-on-demand constraint reduces security of the system by increasing the attack surface on the client side.

To be strictly RESTful and follow all the constraints as they were originally defined, a system should neither deploy authentication nor store session identifiers in HTTP cookies or headers. Since only the absence of security mechanisms allows an entity to provide truly RESTful APIs, a bank claiming to have RESTful APIs either has serious security problems or the APIs do not satisfy all the RESTful requirements.

An important finding is that the concept of RESTful security is impossible. We conclude that the strict REST style on one side and security mechanisms and security best practices on the other side are incompatible. We suggest that secure applications trying to adhere to the REST style should never be called RESTful, but REST-like, i.e. partially adhering to the REST style constraints. Although the term REST-like does appear in some security specifications, such as OpenID Connect [112], it has never been justified from a security perspective.

7.4.2 What to Do

The right security approach is system-specific and heavily dependent on the context. In particular, the frameworks OAuth 2.0 and OpenID Connect rely on TLS for confidentiality, integrity, and server authentication. These frameworks prioritize scalability over security because they use server signed tokens for client authentication. The overall conclusion from the analysis is that systems with high security requirements should deploy client signatures, even though it comes with the cost of

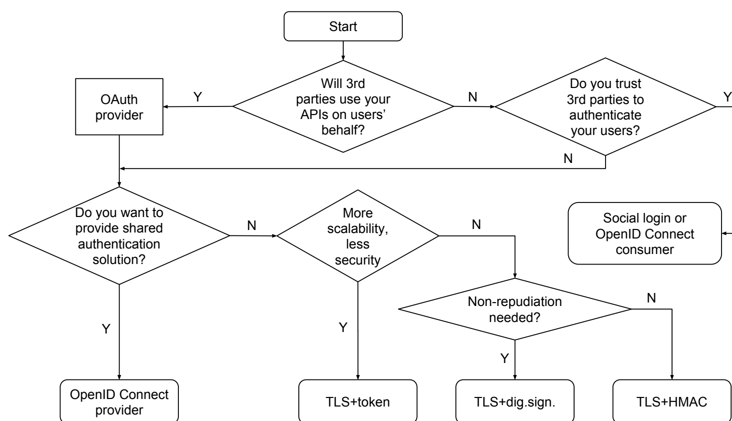


FIGURE 7.2: *Making the right security decision.*

reduced performance when compared to token-based approaches. Social login solutions are both easy to support and convenient for users, but should be avoided if privacy is a serious concern. OAuth should not be relied on for authentication and needs to be combined with a component for authentication. Figure 7.2 contains a flow chart showing how to choose the correct security architecture.

7.4.3 Future Research

Inoue et. al. [53] introduced an architectural style called RESTUS, which incorporates session state at the server-side as an additional constraint. RESTUS partially addresses the security issues of the stateless resource constraint, but not the issues related to the cache and code-on-demand constraints. Similarly to REST, it does not accommodate security. Future research should therefore concentrate on resolving the existing conflicts. A natural progression of this work is to propose an architectural style that incorporates basic security principles.

References

[2] Amazon S3. *Authenticating Requests (AWS Signature v4).*
URL: https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html.

[12] M. Cavage and M. Sporny.
IETF draft. Signing HTTP Messages. (draft-cavage-http-signatures-10). 2018.
URL: https://www.ietf.org/id/draft-cavage-http-signatures-10.txt.

[13] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague.
"OAuth Demystified for Mobile Application Developers".
In: *ACM SIGSAC Conference on Computer and Communications Security.*

Scottsdale, Arizona, USA: ACM, 2014, pp. 892–903.
DOI: 10.1145/2660267.2660323.

[18] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. "One-time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens". In: *ACM Trans. Internet Technol.* 12.1 (July 2012), 1:1–1:24. ISSN: 1533-5399.

[21] F. De Backere, B. Hanssens, R. Heynssens, R. Houthooft, A. Zuliani, S. Verstichel, B. Dhoedt, and F. De Turck. "Design of a security mechanism for RESTful Web Service communication through mobile clients". In: *IEEE Network Operations and Management Symposium*. Krakow, Poland: IEEE, 2014, pp. 1–6. DOI: 10.1109/NOMS.2014.6838308.

[22] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. "Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web". In: *21st USENIX Security Symposium*. Bellevue, WA: USENIX, 2012, pp. 317–331.

[32] C. Fetzer. "Building Critical Applications Using Microservices". In: *IEEE Security Privacy* 14.6 (Nov. 2016), pp. 86–89. ISSN: 1540-7993. DOI: 10.1109/MSP.2016.129.

[33] R. T. Fielding and R. N. Taylor. "Principled Design of the Modern Web Architecture". In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. June 2000, pp. 407–416. DOI: 10.1145/337180.337228.

[35] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000. ISBN: 0-599-87118-0.

[39] K. Fu, E. Sit, K. Smith, and N. Feamster. "The Dos and Don'ts of Client Authentication on the Web." In: *USENIX Security Symposium*. 2001, pp. 251–268.

[40] P. Gorski, L. Lo Iacono, H. Nguyen, and D. Torkian. "Service Security Revisited". In: *IEEE International Conference on Services Computing*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 464–471. DOI: 10.1109/SCC.2014.68.

[44] E. Hammer-Lahav. *RFC 5849. The OAuth 1.0 Protocol*. 2010. DOI: 10.17487/RFC5849.

[53] T. Inoue, H. Asakura, H. Sato, and N. Takahashi. "Key roles of session state: Not against REST architectural style". In: *IEEE 34th Computer Software and Applications Conference*. IEEE. 2010, pp. 171–178.

[55] M. Jones, J. Bradley, and N. Sakimura. *RFC 7519. JSON Web Token*. 2015. DOI: 10.17487/RFC7519.

[66] L. Lo Iacono and H. Nguyen. "Authentication Scheme for REST". In: *International Conference on Future Network Systems and Security*. 2015, pp. 113–128. DOI: 10.1007/978-3-319-19210-9_8.

[74] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 978-0849385230.

[75] Microsoft Azure. *Authentication for the Azure Storage Services*. 2015. URL: https://msdn.microsoft.com/en-us/library/dd179428.aspx.

[100] C. Pautasso, O. Zimmermann, and F. Leymann. "RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision". In: *Proceedings of the 17th International Conference on World Wide Web (WWW'09)*. Beijing, China, 2008, pp. 805–814. DOI: 10.1145/1367497.1367606.

[108] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, 2007. ISBN: 9780596529260.

[112] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. *OpenID Connect Core 1.0*. 2014.

[115] G. Serme, A. de Oliveira, J. Massiera, and Y. Roudier. "Enabling Message Security for RESTful Services". In: *IEEE 19th International Conference on Web Services*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 114–121. DOI: 10.1109/ICWS.2012.94.

[122] S.-T. Sun and K. Beznosov. "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems". In: *ACM Conference on Computer and Communications Security*. Raleigh, North Carolina, USA: ACM, 2012, pp. 378–390. DOI: 10.1145/2382196.2382238.

[127] Trustworthy Internet Movement. *SSL Pulse*. 2017. URL: https://www.trustworthyinternet.org/ssl-pulse/.

[130] R. Wang, S. Chen, and X. Wang. "Signing Me Onto Your Accounts Through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services". In: *IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 365–379. DOI: 10.1109/SP.2012.30.

[131] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization". In: *22nd USENIX Security Symposium*. Washington, DC, 2013, pp. 399–314. ISBN: 978-1-931971-03-4.

Conclusion and Future Work

Answering RQ1. *a) Do microservices have security concerns distinct from those of SOA and distributed systems? b) If so, what is the overlap between microservices and SOA/distributed systems security? What can be reused?*

Conceptually, there is little difference between SOA and microservices. However, there is a practical difference in development and deployment approaches. Security concerns in microservices are indeed similar to those in SOA and distributed systems. Paper I shows that although there are many similarities, SOA and SOA security are generally more complex than what microservices aim for. Microservices are positioned as a lightweight and easy to start with architecture; microservice security should ideally follow the same principles.

As explained in Section 2.2.4, SOA security comprises of general security standards and multiple Web Services security standards, including XML security standards. However, microservices are not limited to web services and XML format and can employ various communication protocols. Much of SOA security can be reused for microservices given enough harmonization and porting effort, but may feel out of place or be too complicated for the average microservice developer.

These results led us to look deeper into specific advantages and disadvantages of microservice architecture, resulting in research questions number 2 and 3.

Answering RQ2. *What are the security challenges on the path to microservice adoption? How can these challenges be addressed?*

There are several key security challenges on the path to microservice adoption:

- *Automation of microservice security solutions* on all levels is of critical importance because manual security provisioning and manual intrusion response are infeasible on a large scale. Microservice networks consisting of hundreds or thousands of nodes require automated security mechanisms that scale well. There is a need for tools and frameworks to improve microservice security, ideally with open source code. We attempt to address this issue by developing the MiSSFire framework to enforce MTLS and security tokens with a self-hosted PKI (Paper I), and designing the μGE intrusion response system with a game theoretic approach (Paper III).

- *Microservice security guidelines and standards* are currently missing and represent a security challenge of its own. Formalization of microservice security

principles and developers awareness of them is the first step towards more secure microservice implementations. Such standards should be harmonized with and reuse the existing and well-known security mechanisms and protocols such as TLS and OAuth. However, the pitfalls of SOA and WS-* security stack, namely unmanageable complexity, should be avoided. The issue of microservice security standards is not addressed in this thesis due to the global nature of the problem; it requires community effort.

Answering RQ3. *Can the microservice architectural style lead to better security? If so, what are the opportunities enabled by it?*

Microservice architecture can lead to better security because it facilitates defense in depth by design, as shown in Paper I, and has inherent benefits of increased service isolation and software diversity, as shown in Paper II. However, it is always possible to build insecure systems independently of what architecture is adopted: microservices are in no way a security panacea.

Open questions and future work. Plenty of open questions remain on the topic of microservice security. Some open questions raised in the Papers I - IV are: to what extent can arbitrary microservices benefit from hardening and diversification; what security overhead is industry willing to accept; can neural networks be used for microservice IRS; what is the best authorization solution for inter-service communication; is automated splitting and merging of microservices a feasible and effective intrusion response.

Moreover, there is a lack of real-world statistically significant data on how production-ready microservice networks are structured and how the attacks spread in them. Such datasets are needed for testing and effectiveness evaluation of the future security tools on simulated microservice networks, e.g. evolving our automated response system.

We believe it would be fruitful to design tools for automated microservice diversification and evaluate the effectiveness of the proposed IRS for real-world microservice networks.

Bibliography

[1] N. Alshuqayran, N. Ali, and R. Evans. "A Systematic Mapping Study in Microservice Architecture". In: *Service-Oriented Computing and Applications (SOCA 2016)*. Nov. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15 (cit. on p. 54).

[2] Amazon S3. *Authenticating Requests (AWS Signature v4)*. URL: https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html (cit. on p. 112).

[3] Amazon Web Services, Inc. *AWS Lambda Pricing*. URL: https://aws.amazon.com/lambda/pricing/ (visited on May 21, 2018) (cit. on p. 27).

[4] J. Armstrong. "Making Reliable Distributed Systems in the Presence of Software Errors". PhD thesis. Stockholm, Sweden: The Royal Institute of Technology, Dec. 2003 (cit. on p. 58).

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33 (cit. on p. 102).

[6] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *W3C Recommendation. XML Signature v1.1*. 2013 (cit. on p. 33).

[7] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. "Chaos engineering". In: *IEEE Software* 33.3 (2016), pp. 35–41 (cit. on p. 24).

[8] N. Bhalla. "Web services vulnerabilities". In: *Black Hat Europe*. USENIX Association, 2007. URL: https://www.blackhat.com/presentations/bh-europe-07/Bhalla-Kazerooni/Whitepaper/bh-eu-07-bhalla-WP.pdf (cit. on p. 34).

[9] K. Birman. "The promise, and limitations, of gossip protocols". In: *ACM SIGOPS Operating Systems Review* 41.5 (2007), pp. 8–13 (cit. on p. 42).

[10] L. C. Briand, J. W. Daly, and J. K. Wust. "A unified framework for coupling measurement in object-oriented systems". In: *IEEE Transactions on Software Engineering* 25.1 (1999), pp. 91–121 (cit. on p. 38).

[11] O. Catakoglu, M. Balduzzi, and D. Balzarotti. "Attacks landscape in the dark side of the web". In: *Proceedings of the Symposium on Applied Computing*. ACM. 2017, pp. 1739–1746 (cit. on p. 35).

[12] M. Cavage and M. Sporny. *IETF draft. Signing HTTP Messages*. (draft-cavage-http-signatures-10). 2018. URL: https://www.ietf.org/id/draft-cavage-http-signatures-10.txt (cit. on p. 113).

[13] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. "OAuth Demystified for Mobile Application Developers". In: *ACM SIGSAC Conference on Computer and Communications Security*. Scottsdale, Arizona, USA: ACM, 2014, pp. 892–903. DOI: 10.1145/2660267.2660323 (cit. on pp. 32, 113).

[14] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull. "Exploring Language Support for Immutability". In: *Int'l Conf. on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 736–747. DOI: 10.1145/2884781.2884798 (cit. on p. 62).

[15] T. Combe, A. Martin, and R. Di Pietro. "To Docker or not to Docker: A security perspective". In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62 (cit. on p. 60).

[16] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. *RFC 5280. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. 2008. DOI: 10.17487/RFC5280 (cit. on p. 30).

[17] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. *N-Variant Systems A Secretless Framework for Security through Diversity*. 2006 (cit. on p. 86).

[18] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. "One-time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens". In: *ACM Trans. Internet Technol.* 12.1 (July 2012), 1:1–1:24. ISSN: 1533-5399 (cit. on p. 112).

[19] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. "Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM". In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS '13. Hangzhou, China: ACM, 2013, pp. 299–310. DOI: 10.1145/2484313.2484351 (cit. on p. 85).

[20] C. Davis. "What if the Web Were Not RESTful?" In: *Proceedings of the Third International Workshop on RESTful Design*. WS-REST '12. Lyon, France: ACM, 2012, pp. 3–10. DOI: 10.1145/2307819.2307823 (cit. on p. 37).

[21] F. De Backere, B. Hanssens, R. Heynssens, R. Houthooft, A. Zuliani, S. Verstichel, B. Dhoedt, and F. De Turck. "Design of a security mechanism for RESTful Web Service communication through mobile clients". In: *IEEE Network Operations and Management Symposium*. Krakow, Poland: IEEE, 2014, pp. 1–6. DOI: 10.1109/NOMS.2014.6838308 (cit. on p. 110).

[22] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. "Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web". In: *21st USENIX Security Symposium*. Bellevue, WA: USENIX, 2012, pp. 317–331 (cit. on p. 112).

[23] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654 (cit. on p. 29).

[24] P. Dobbelaere and K. S. Esmaili. "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper". In: *DEBS*. ACM. 2017, pp. 227–238 (cit. on p. 41).

[25] B. Doerrfeld. *How To Control User Identity Within Microservices*. Nordic APIs blog, 2016. URL: https://nordicapis.com/how-to-control-user-identity-within-microservices/ (visited on Nov. 1, 2017) (cit. on p. 65).

[26] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. "Microservices: yesterday, today, and tomorrow". In: *CoRR* abs/1606.04036 (2016) (cit. on pp. 54, 57, 62, 80).

[27] C. M. Ellison, B. Frantz, B. Thomas, T. Ylonen, R. L. Rivest, and B. Lampson. *SPKI certificate theory*. IETF RFC 2693, 1999. URL: https://tools.ietf.org/html/rfc2693 (visited on Nov. 1, 2017) (cit. on p. 67).

[28] C. Ellison and B. Schneier. "Ten risks of PKI: What you're not being told about public key infrastructure". In: *Comput Secur J* 16.1 (2000), pp. 1–7 (cit. on p. 30).

[29] R. A. van Engelen and W. Zhang. "An Overview and Evaluation of Web Services Security Performance Optimizations". In: *Proceedings of the 2008 IEEE International Conference on Web Services*. ICWS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 137–144. DOI: 10.1109/ICWS.2008.102 (cit. on p. 34).

[30] J. R. Erenkrantz, M. Gorlick, G. Suryanarayana, and R. N. Taylor. "From representations to computations: the evolution of web architectures". In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2007, pp. 255–264 (cit. on p. 37).

[31] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. "The many faces of publish/subscribe". In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131 (cit. on pp. 38, 39).

[32] C. Fetzer. "Building Critical Applications Using Microservices". In: *IEEE Security Privacy* 14.6 (Nov. 2016), pp. 86–89. ISSN: 1540-7993. DOI: 10.1109/MSP.2016.129 (cit. on pp. 54, 62, 80, 98, 116).

[33] R. T. Fielding and R. N. Taylor. "Principled Design of the Modern Web Architecture". In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. June 2000, pp. 407–416. DOI: 10.1145/337180.337228 (cit. on pp. 36, 37, 114).

[34] R. T. Fielding, R. N. Taylor, J. R. Erenkrantz, M. M. Gorlick, J. Whitehead, R. Khare, and P. Oreizy. "Reflections on the REST architectural style and principled design of the modern web architecture (impact paper award)". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 4–14 (cit. on p. 37).

[35] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000. ISBN: 0-599-87118-0 (cit. on pp. 109, 114–117).

[36] G. Folino and P. Sabatino. "Ensemble Based Collaborative and Distributed Intrusion Detection Systems". In: *J. Netw. Comput. Appl.* 66.C (May 2016), pp. 1–16. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2016.03.011 (cit. on p. 101).

[37] M. Fowler. *Microservice Trade-Offs*. July 2015. URL: http://martinfowler.com/articles/microservice-trade-offs.html (visited on Apr. 13, 2016) (cit. on p. 79).

[38] M. Fowler and J. Lewis. *Microservices*. Mar. 2014. URL: http://www.martinfowler.com/articles/microservices.html (visited on Nov. 1, 2017) (cit. on pp. 19, 40, 54, 55, 57, 62).

[39] K. Fu, E. Sit, K. Smith, and N. Feamster. "The Dos and Don'ts of Client Authentication on the Web." In: *USENIX Security Symposium*. 2001, pp. 251–268 (cit. on pp. 65, 112, 115).

[40] P. Gorski, L. Lo Iacono, H. Nguyen, and D. Torkian. "Service Security Revisited". In: *IEEE International Conference on Services Computing*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 464–471. DOI: 10.1109/SCC.2014.68 (cit. on p. 110).

[41] S. Graham, D. Davis, S. Simeonov, G. Daniels, P. Brittenham, Y. Nakamura, P. Fremantle, D. Konig, and C. Zentner. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. Sams Publishing, 2005. ISBN: 0-672-32641-8 (cit. on p. 33).

[42] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. *SOAP Version 1.2 Part 1: Messaging Framework*. 2007. URL: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/ (cit. on p. 25).

[43] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann. "Service cutter: A systematic approach to service decomposition". In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2016, pp. 185–200 (cit. on p. 26).

[44] E. Hammer-Lahav. *RFC 5849. The OAuth 1.0 Protocol*. 2010. DOI: 10.17487/RFC5849 (cit. on pp. 30, 113).

[45] D. Hardt. *RFC 6749. The OAuth 2.0 Authorization Framework*. 2012. DOI: 10.17487/RFC6749 (cit. on p. 31).

[46] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. "Gremlin: systematic resilience testing of microservices". In: *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE. 2016, pp. 57–66 (cit. on p. 24).

[47] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson. "ILR: Where'd My Gadgets Go?" In: *Security and Privacy (SP), 2012 IEEE Symposium on*. May 2012, pp. 571–585. DOI: 10.1109/SP.2012.39 (cit. on p. 85).

[48] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2003. ISBN: 0321200683 (cit. on p. 36).

[49] J. Holgersson and E. Söderström. "Web service security – vulnerabilities and threats within the context of WS-security". In: *4th IEEE Conference on Standardization and Innovation in Information Technology (SIIT 2005), 21-23 September 2005, Geneva, Switzerland*. IEEE, 2005, pp. 138–146. DOI: 10.1109/SIIT.2005.1563803 (cit. on p. 34).

[50] P. Hu, R. Yang, Y. Li, and W. C. Lau. "Application Impersonation: Problems of OAuth and API Design in Online Social Networks". In: *2nd ACM Conference on Online Social Networks*. Dublin, Ireland: ACM, 2014, pp. 271–278. DOI: 10.1145/2660460.2660463 (cit. on p. 32).

[51] T. Imamura, B. Dillaway, E. Simon, K. Yiu, and M. Nyström. *W3C Recommendation. XML Encryption v1.1*. 2013 (cit. on p. 33).

[52] 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) (cit. on p. 16).

[53] T. Inoue, H. Asakura, H. Sato, and N. Takahashi. "Key roles of session state: Not against REST architectural style". In: *IEEE 34th Computer Software and Applications Conference*. IEEE. 2010, pp. 171–178 (cit. on pp. 37, 110, 116, 119).

[54] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. "Compiler-Generated Software Diversity". In: *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Ed. by S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang. New York, NY: Springer New York, 2011, pp. 77–98. DOI: 10.1007/978-1-4614-0977-9_4 (cit. on pp. 86, 102).

[55] M. Jones, J. Bradley, and N. Sakimura. *RFC 7519. JSON Web Token*. 2015. DOI: 10.17487/RFC7519 (cit. on p. 111).

[56] A. Jøsang. "A Consistent Definition of Authorization". In: *International Workshop on Security and Trust Management*. Springer. 2017, pp. 134–144 (cit. on p. 28).

[57] N. Josuttis. *SOA in Practice: The Art of Distributed System Design*. Sebastopol: O'Reilly Media, Inc., 2007. ISBN: 0596529554 (cit. on pp. 25, 26, 56).

[58] R. Khare and R. N. Taylor. "Extending the representational state transfer (REST) architectural style for decentralized systems". In: *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*. IEEE. 2004, pp. 428–437. DOI: 10.1109/ICSE.2004.1317465 (cit. on p. 37).

[59] S. Kim, F. B. Bastani, I.-L. Yen, and R. Chen. "High-assurance synthesis of security services from basic microservices". In: *Software Reliability Engineering (ISSRE 2003)*. IEEE. 2003, pp. 154–165 (cit. on p. 56).

[60] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01203 (cit. on p. 59).

[61] D. Kupser, C. Mainka, J. Schwenk, and J. Somorovsky. "How to Break XML Encryption – Automatically". In: *9th USENIX Workshop on Offensive Technologies*. Washington, DC: USENIX Association, 2015. URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/kupser (cit. on p. 34).

[62] P. Laplante. *What Every Engineer Should Know about Software Engineering*. What Every Engineer Should Know. CRC Press, 2007. ISBN: 9781420006742 (cit. on p. 20).

[63] F. Lascelles and A. Flint.
WS Security Performance: Secure Conversation versus the X509 Profile. Apr. 2006.
URL: http://francoislascelles.sys-con.com/node/204424/ (cit. on p. 34).

[64] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. "The highways and country roads to continuous deployment". In: *IEEE Software* 32.2 (2015), pp. 64–72 (cit. on p. 20).

[65] LightStep, Inc. *The 2018 Global Microservices Trends report*. Apr. 2018.
URL: https://go.lightstep.com/global-microservices-trends-report-2018 (visited on May 21, 2018) (cit. on p. 16).

[66] L. Lo Iacono and H. Nguyen. "Authentication Scheme for REST".
In: *International Conference on Future Network Systems and Security*. 2015, pp. 113–128. DOI: 10.1007/978-3-319-19210-9_8 (cit. on pp. 110, 112).

[67] O. Lysne, K. J. Hole, C. Otterstad, Ø. Ytrehus, R. Aarseth, and J. Tellnes.
"Vendor Malware: Detection Limits and Mitigation".
In: *Computer* 49.8 (Aug. 2016), pp. 62–69. ISSN: 0018-9162.
DOI: 10.1109/MC.2016.227 (cit. on pp. 80, 82).

[68] A. Madhavapeddy and D. J. Scott.
"Unikernels: Rise of the Virtual Library Operating System".
In: *Queue* 11.11 (Dec. 2013), 30:30–30:44. ISSN: 1542-7730.
DOI: 10.1145/2557963.2566628 (cit. on p. 58).

[69] A. L. Marcon, A. O. Santin, L. A. de Paula Lima, R. R. Obelheiro, and M. Stihler.
"Policy control management for Web Services".
In: *Integrated Network Management (IM'09)*. IEEE. 2009, pp. 49–56 (cit. on p. 67).

[70] V. Mavroudis, A. Cerulli, P. Svenda, D. Cvrcek, D. Klinec, and G. Danezis.
"A Touch of Evil: High-Assurance Cryptographic Hardware from Untrusted Components". In: *Computer and Communications Security (CCS'2017)*.
ACM, 2017, pp. 1583–1600. DOI: 10.1145/3133956.3133961 (cit. on p. 60).

[71] J. McHugh, A. Christie, and J. Allen.
"Defending yourself: The role of intrusion detection systems".
In: *IEEE Software* 17.5 (2000), pp. 42–51 (cit. on p. 35).

[72] M. D. McIlroy. "Mass produced software components". In: *Software Engineering: Report of a conference sponsored by the NATO Science Committee*.
Garmisch, Germany: Scientific Affairs Division, NATO, Jan. 1969, pp. 79–87.
URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF (cit. on p. 20).

[73] M. McIntosh and P. Austel.
"XML Signature Element Wrapping Attacks and Countermeasures".
In: *Workshop on Secure Web Services*. Fairfax, VA, USA: ACM, 2005, pp. 20–27.
DOI: 10.1145/1103022.1103026 (cit. on p. 34).

[74] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 978-0849385230 (cit. on p. 111).

[75] Microsoft Azure. *Authentication for the Azure Storage Services*. 2015. URL: https://msdn.microsoft.com/en-us/library/dd179428.aspx (cit. on p. 112).

[76] Microsoft Azure. *Introducing Azure confidential computing*. Sept. 2017. URL: https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/ (visited on Nov. 1, 2017) (cit. on p. 71).

[77] V. Mladenov, C. Mainka, J. Krautwald, F. Feldmann, and J. Schwenk. "On the security of modern Single Sign-On Protocols: OpenID Connect 1.0". In: *CoRR* (2015) (cit. on p. 33).

[78] B. Möller, T. Duong, and K. Kotowicz. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. Sept. 2014. URL: https://www.openssl.org/~bodo/ssl-poodle.pdf (visited on Nov. 1, 2017) (cit. on p. 60).

[79] D. Mónica. *MTLS in a Microservices World*. Talk at BSides Lisbon 2016. Nov. 2016. URL: https://www.youtube.com/watch?v=apma_C24W58 (visited on July 1, 2018) (cit. on p. 63).

[80] F. Montesi and J. Weber. "Circuit Breakers, Discovery, and API Gateways in Microservices". In: *CoRR* abs/1609.05830v2 (2016). URL: http://arxiv.org/abs/1609.05830v2 (cit. on pp. 24, 60, 63, 83).

[81] A. Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, 2016. ISBN: 9781491915769 (cit. on p. 22).

[82] S. Murer and C. Hagen. "Fifteen years of service-oriented architecture at Credit Suisse". In: *IEEE Software* 31.6 (2014), pp. 9–15 (cit. on p. 25).

[83] A. Nadalin, M. Goodner, M. Gudgin, D. Turner, A. Barbir, and H. Granqvist. *OASIS Standard Specification. WS-Trust 1.4*. Apr. 2012. URL: http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html (visited on Nov. 1, 2017) (cit. on p. 65).

[84] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. *OASIS Standard Specification. WS-Security*. 2006 (cit. on p. 33).

[85] S. Newman. *Building Microservices*. O'Reilly Media, 2015. ISBN: 9781491950357 (cit. on pp. 19, 24, 26, 36, 38, 54, 55, 57, 61, 62, 79, 80, 97).

[86] NGINX, Inc. *The Future of Application Development and Delivery Is Now*. Nov. 2015. URL: https://www.nginx.com/resources/library/app-dev-survey/ (visited on Jan. 6, 2017) (cit. on pp. 16, 79).

[87] N. Nordbotten. "XML and Web Services Security Standards". Undetermined. In: *IEEE Communications Surveys and Tutorials* 11.3 (2009), pp. 4–21. DOI: 10.1109/SURV.2009.090302 (cit. on p. 33).

[88] M. A. Noureddine, A. Fawaz, W. H. Sanders, and T. Başar. "A Game-Theoretic Approach to Respond to Attacker Lateral Movement". In: *Proceedings of the 7th International Conference on Decision and Game Theory for Security (GameSec 2016).* Ed. by Q. Zhu, T. Alpcan, E. Panaousis, M. Tambe, and W. Casey. Springer International Publishing, 2016, pp. 294–313. DOI: 10.1007/978-3-319-47413-7_17 (cit. on pp. 98, 102).

[89] M. Nygard. *Release It! Design and Deploy Production-ready Software*. Pragmatic Bookshelf Series. 2007. ISBN: 9780978739218 (cit. on p. 83).

[90] *Official Docker v17.06 documentation. Manage swarm security with public key infrastructure.* URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki (visited on Nov. 1, 2017) (cit. on p. 63).

[91] T. Ormandy. *FireEye Exploitation: Project Zero's Vulnerability of the Beast.* Dec. 2015. URL: https://googleprojectzero.blogspot.no/2015/12/fireeye-exploitation-project-zeros.html (visited on Feb. 7, 2017) (cit. on p. 88).

[92] M. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994. ISBN: 9780262650403 (cit. on p. 98).

[93] C. Otterstad and T. Yarygina. "Low-Level Exploitation Mitigation by Diverse Microservices". In: *Service-Oriented and Cloud Computing (ESOCC 2017).* Ed. by F. De Paoli, S. Schulte, and E. Broch Johnsen. Oslo, Norway: Springer, 2017, pp. 49–56. DOI: 10.1007/978-3-319-67262-5_4 (cit. on pp. 54, 62, 63, 77, 98, 100, 102).

[94] *OWASP Top 10 – 2017: The Ten Most Critical Web Application Security Risks.* 2017. URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%5C%28en%5C%29.pdf.pdf (visited on Nov. 1, 2017) (cit. on p. 60).

[95] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 2nd. Springer-Verlag Berlin Heidelberg, 2010. ISBN: 978-3-642-04100-6 (cit. on p. 29).

[96] D. Partridge and W. Krzanowski. "Software diversity: practical statistics for its measurement and exploitation". In: *Information and Software Technology* 39.10 (1997), pp. 707–717. ISSN: 0950-5849. DOI: 10.1016/S0950-5849(97)00023-2 (cit. on p. 85).

[97] C. Pautasso and E. Wilde. "Why is the web loosely coupled?: A multi-faceted metric for service design". In: *Proceedings of the 18th International Conference on World Wide Web (WWW'09).* ACM. 2009, pp. 911–920. DOI: 10.1145/1526709.1526832 (cit. on p. 38).

[98] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. "Microservices in Practice, Part 1: Reality Check and Service Design". In: *IEEE Software* 34.1 (Jan.–Feb. 2017), pp. 91–98. DOI: 10.1109/MS.2017.24 (cit. on pp. 54, 62, 79, 97).

[99] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. "Microservices in Practice, Part 2: Service Integration and Sustainability". In: *IEEE Software* 34.2 (Mar.–Apr. 2017), pp. 97–104. ISSN: 0740-7459. DOI: 10.1109/MS.2017.56 (cit. on pp. 38, 57, 62).

[100] C. Pautasso, O. Zimmermann, and F. Leymann. "RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision". In: *Proceedings of the 17th International Conference on World Wide Web (WWW'09)*. Beijing, China, 2008, pp. 805–814. DOI: 10.1145/1367497.1367606 (cit. on pp. 37, 60, 109).

[101] B. Payne. *PKI at Scale Using Short-lived Certificates*. Talk at USENIX Enigma 2016. URL: https://youtu.be/7YPlsbz8Pig (visited on July 1, 2018) (cit. on p. 64).

[102] N. Provos. "A Virtual Honeypot Framework". In: *USENIX Security Symposium*. Vol. 173. 2004, pp. 1–14 (cit. on p. 35).

[103] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. "Flip Feng Shui: Hammering a Needle in the Software Stack". In: *25th USENIX Security Symposium*. USENIX, Aug. 2016, pp. 1–18. ISBN: 978-1-931971-32-4 (cit. on p. 60).

[104] D. Renzel, P. Schlebusch, and R. Klamma. "Today's top "RESTful" services and why they are not RESTful". In: *International Conference on Web Information Systems Engineering*. Springer. 2012, pp. 354–367 (cit. on p. 37).

[105] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. (draft-ietf-tls-tls13-28). (to be approved as RFC 8446). 2018. URL: https://www.rfc-editor.org/internet-drafts/draft-ietf-tls-tls13-28.txt (visited on July 1, 2018) (cit. on p. 29).

[106] M. Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, 2015. ISBN: 978-1-491-94161-4 (cit. on pp. 57, 62).

[107] C. Richardson and F. Smith. *Microservices: From Design to Deployment*. NGINX, Inc., 2016. URL: https://www.nginx.com/resources/library/designing-deploying-microservices/ (cit. on pp. 26, 36, 60, 62, 79, 83).

[108] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, 2007. ISBN: 9780596529260 (cit. on pp. 109, 115).

[109] R. L. Rivest. "Can We Eliminate Certificate Revocation Lists?" In: *In Financial Cryptography*. Springer-Verlag, 1998, pp. 178–183 (cit. on p. 64).

[110] R. L. Rivest and B. Lampson. *SDSI—a simple distributed security infrastructure*. MIT. 1996. URL: https://people.csail.mit.edu/rivest/sdsi10.html (cit. on p. 67).

[111] S. Roy, C. Ellis, S. Shiva, D. Dasgupta, V. Shandilya, and Q. Wu. "A survey of game theory as applied to network security". In: *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*. IEEE. 2010, pp. 1–10 (cit. on p. 98).

[112] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. *OpenID Connect Core 1.0*. 2014 (cit. on pp. 33, 65, 113, 118).

[113] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina. "The Halting Problems of Network Stack Insecurity". In: *;login:* 36.6 (Dec. 2011) (cit. on pp. 84, 89).

[114] R. W. Schulte and Y. V. Natis. *"Service Oriented" Architectures, Part 1*. Apr. 1996. URL: https://www.gartner.com/doc/302868/ (visited on May 21, 2018) (cit. on p. 25).

[115] G. Serme, A. de Oliveira, J. Massiera, and Y. Roudier. "Enabling Message Security for RESTful Services". In: *IEEE 19th International Conference on Web Services*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 114–121. DOI: 10.1109/ICWS.2012.94 (cit. on p. 110).

[116] H. Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313 (cit. on p. 81).

[117] Y. Sheffer, R. Holz, and P. Saint-Andre. *RFC 7457. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*. 2015. DOI: 10.17487/RFC7457 (cit. on p. 29).

[118] C. B. Simmons, S. G. Shiva, H. S. Bedi, and V. Shandilya. "ADAPT: a game inspired attack-defense and performance metric taxonomy". In: *IFIP International Information Security Conference*. Springer. 2013, pp. 344–365 (cit. on p. 103).

[119] A. Singhal, T. Winograd, and K. Scarfone. *Guide to Secure Web Services*. 2007. URL: http://csrc.nist.gov/publications/nistpubs/800-95/SP800-95.pdf (visited on Dec. 1, 2015) (cit. on p. 33).

[120] *Snort official web-site*. URL: https://www.snort.org (visited on Feb. 23, 2018) (cit. on p. 101).

[121] N. Stakhanova, S. Basu, and J. Wong. "A taxonomy of intrusion response systems". In: *International Journal of Information and Computer Security* 1.1-2 (2007), pp. 169–184 (cit. on pp. 35, 98).

[122] S.-T. Sun and K. Beznosov. "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems". In: *ACM Conference on Computer and Communications Security*. Raleigh, North Carolina, USA: ACM, 2012, pp. 378–390. DOI: 10.1145/2382196.2382238 (cit. on pp. 32, 113).

[123] Y. Sun, S. Nanda, and T. Jaeger. "Security-as-a-service for microservices-based cloud applications". In: *Cloud Computing Technology and Science (CloudCom 2015)*. IEEE. 2015, pp. 50–57 (cit. on pp. 54, 98).

[124] H. Takabi, J. B. D. Joshi, and G. J. Ahn. "Security and Privacy Challenges in Cloud Computing Environments". In: *IEEE Security Privacy* 8.6 (Nov. 2010), pp. 24–31. ISSN: 1540-7993. DOI: 10.1109/MSP.2010.186 (cit. on p. 60).

[125] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007. ISBN: 9780132392273 (cit. on pp. 22–24, 58, 63, 97).

[126] *The Heartbleed bug in OpenSSL library*. Apr. 2014. URL: http://heartbleed.com/ (visited on Nov. 1, 2017) (cit. on p. 60).

[127] Trustworthy Internet Movement. *SSL Pulse*. 2017. URL: https://www.trustworthyinternet.org/ssl-pulse/ (cit. on p. 117).

[128] T. Ueda, T. Nakaike, and M. Ohara. "Workload characterization for microservices". In: *IISWC 2016*. IEEE. 2016, pp. 85–94 (cit. on p. 70).

[129] S. Vinoski. "Old measures for new services". In: *IEEE Internet Computing* 9.6 (2005), pp. 72–74 (cit. on p. 38).

[130] R. Wang, S. Chen, and X. Wang. "Signing Me Onto Your Accounts Through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services". In: *IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 365–379. DOI: 10.1109/SP.2012.30 (cit. on p. 113).

[131] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization". In: *22nd USENIX Security Symposium*. Washington, DC, 2013, pp. 399–314. ISBN: 978-1-931971-03-4 (cit. on pp. 32, 113).

[132] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 157–168. DOI: 10.1145/2382196.2382216 (cit. on p. 85).

[133] Y. Yarom and K. Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium*. USENIX, 2014, pp. 719–732. ISBN: 978-1-931971-15-7 (cit. on p. 59).

[134] T. Yarygina. "RESTful Is Not Secure". In: *Applications and Techniques in Information Security (ATIS 2017)*. Springer, 2017, pp. 141–153. DOI: 10.1007/978-981-10-5421-1_12 (cit. on p. 60).

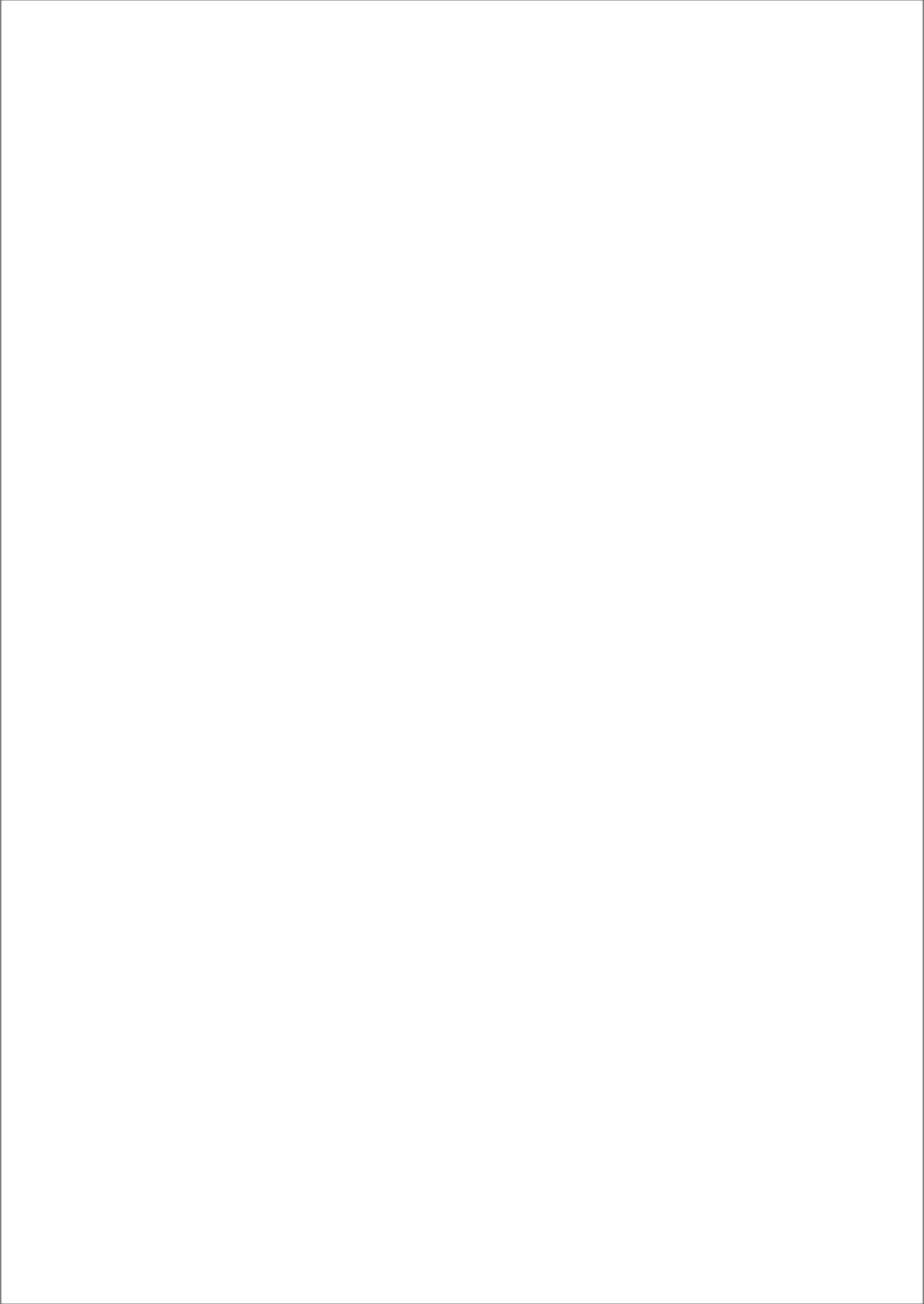
[135] T. Yarygina and A. H. Bagge. "Overcoming Security Challenges in Microservice Architectures". In: *12th IEEE International Symposium on Service-Oriented System Engineering (SOSE'18)*. Bamberg, Germany: IEEE, Mar. 2018, pp. 11–20. DOI: 10.1109/SOSE.2018.00011 (cit. on pp. 51, 98, 100).

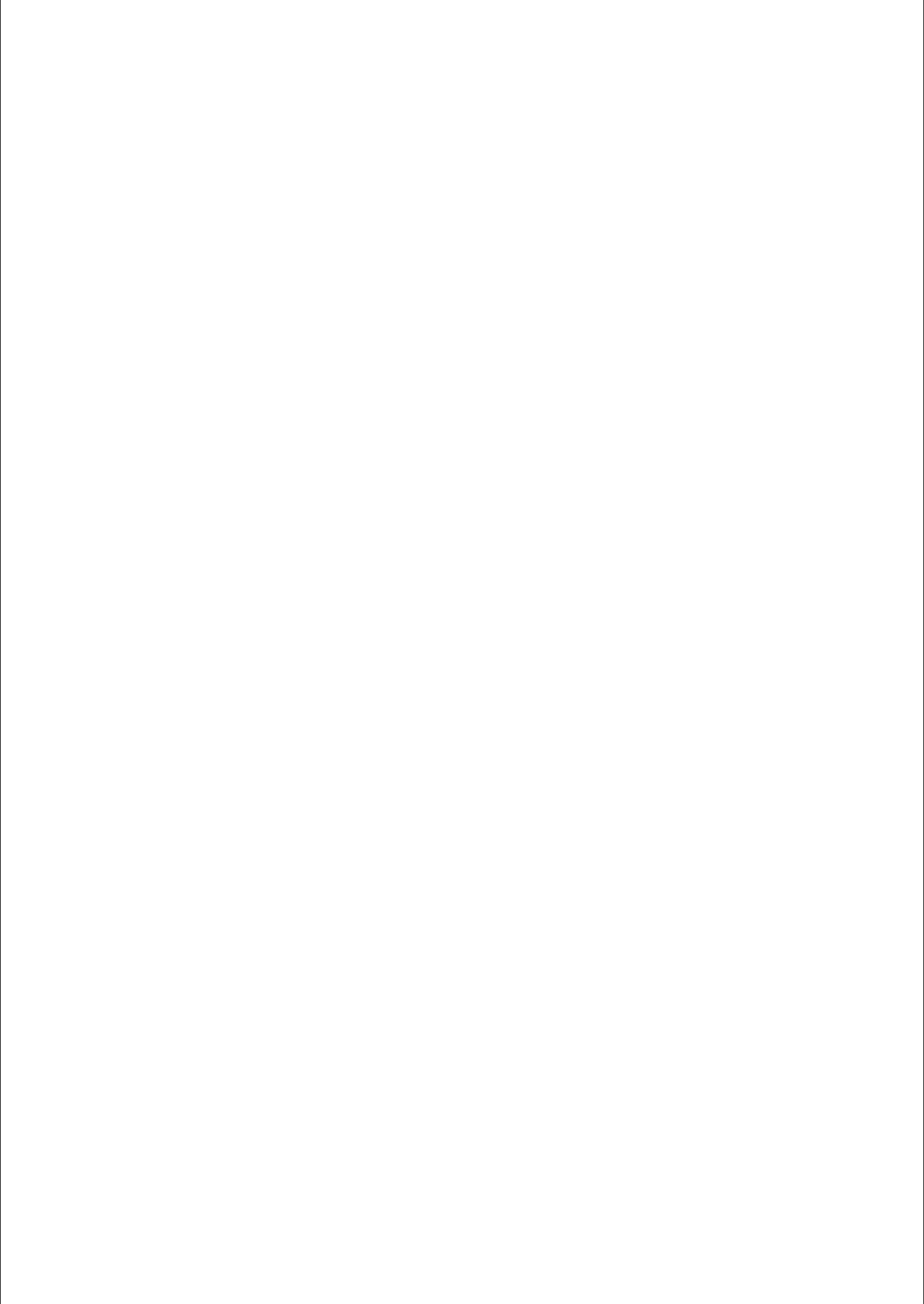
[136] T. Yarygina and C. Otterstad. "A Game of Microservices: Automated Intrusion Response". In: *Distributed Applications and Interoperable Systems*. Ed. by S. Bonomi and E. Rivière. Cham: Springer International Publishing, June 2018, pp. 169–177. DOI: 10.1007/978-3-319-93767-0_12 (cit. on p. 95).

[137] E. Yuan, N. Esfahani, and S. Malek. "A systematic survey of self-protecting software systems". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 8.4 (2014), p. 17 (cit. on p. 98).

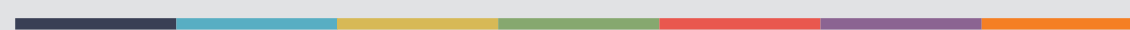
[138] O. Zimmermann. "Microservices tenets: Agile approach to service development and deployment". In: *Computer Science - Research and Development* (2016), pp. 1–10. ISSN: 1865-2042. DOI: 10.1007/s00450-016-0337-0 (cit. on pp. 26, 38, 57, 62, 80, 97).

[139] S. A. Zonouz, H. Khurana, W. H. Sanders, and T. M. Yardley. "RRE: A game-theoretic intrusion response and recovery engine". In: *IEEE Transactions on Parallel and Distributed Systems* 25.2 (2014), pp. 395–406 (cit. on p. 98).





Graphic design: Communication Division, UIB / Print: Skipnes Kommunikasjon AS



uib.no

ISBN: 978-82-308-3665-1