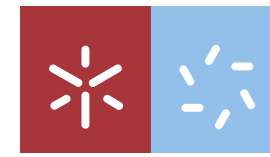




Nelson Diogo Santos Loureiro **Compositional Analysis of Vulnerabilities in Microservice-based Applications**

UMinho | 2021

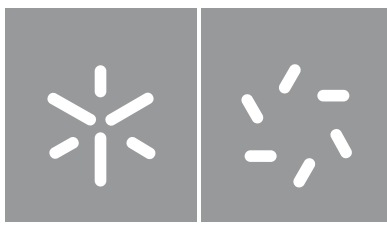


Universidade do Minho
Escola de Ciências

Nelson Diogo Santos Loureiro

**Compositional Analysis
of Vulnerabilities in
Microservice-based Applications**

janeiro de 2021



Universidade do Minho

Escola de Ciências

Nelson Diogo Santos Loureiro

**Compositional Analysis
of Vulnerabilities in
Microservice-based Applications**

Dissertação de Mestrado
em Matemática e Computação

Trabalho efetuado sob a orientação do
Professor Doutor Luís Filipe Ribeiro Pinto
e da
Professora Doutora Maria João Gomes Frade

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Atribuição

CC BY

<https://creativecommons.org/licenses/by/4.0/>

Acknowledgments

First I would like to thank my parents and my brother for all the support they given me this year. Without their encouragement, this dissertation would not have been possible.

I would like to thank my supervisors, Professor Luís Pinto and Professor Maria João Frade, for their help, patience, and guidance throughout this year. I learned a lot from them in this dissertation.

To all my colleagues from Checkmarx. Especially the two who helped me, throughout this year, understand the concepts needed for this work, and to those from my internship that were there to answer my stupid questions.

I also would like to thank all my friends that were there me in the good times and the bad times. Special thanks to the three that gave me motivation to continue on, and complete my academic journey.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Abstract

Nowadays, software applications are becoming larger, more expensive, and more complex due to high market demands and adoption of new technologies that have emerged in recent years. Several companies, in order to remain competitive and solve problems resulting from traditional architectural styles, have started to develop their applications following the microservice architectural style.

Microservices are an approach to distributed systems, where a single application is developed as a collection of small services, each running its own process and communicating over a network using lightweight mechanisms, such as the HTTP protocol.

All types of applications, including those based on microservices, may contain vulnerabilities, which when exploited by a malicious user can cause problems for the organization developing the application and/or the other users. One way to prevent such problems is through the detection and correction of vulnerabilities. However, due to the way microservice-based applications are developed, there exists an additional challenge on how to detect vulnerabilities resulting from service-to-service communications, since these type of interactions are entirely different from those that occur within traditional applications.

In collaboration with Checkmarx, we developed a solution that detects vulnerabilities resulting from service-to-service interactions. This was achieved with the help of CxSAST, a static analyzer of Checkmarx built to discover vulnerabilities in source code, and CxQL queries, which implement a technique of data-flow analysis. We applied a compositional approach, which analyses one service at a time, and then connects the results. Our work is directed at services that communicate through the HTTP protocol and interact in a way that satisfies the REST architectural style.

Our solution consists of a collection of queries and of an external tool. The queries produce results, searching for traces of vulnerabilities in the source code of services. The external tool, implemented in the C# language, connects these results and defines paths between them. This is done with the use of an adapted depth-first search algorithm over a graph where each vertex is a result, or a combination of results produced by the queries. In order to only provide relevant information, some decisions have been taken about the type of paths to be produced by the external tool, such as the decision to avoid subpaths, which in this context are a source of redundancy.

Keywords: Data-flow analysis, Graphs, Microservices, Static analysis, Vulnerabilities.

Resumo

Atualmente, as aplicações de *software* estão a tornar-se maiores, mais dispendiosas e mais complexas devido à elevada exigência do mercado e da adoção de novas tecnologias que têm surgido nos últimos anos. Várias empresas, a fim de se manterem competitivas e resolverem problemas resultantes dos estilos arquiteturais tradicionais, começaram a desenvolver as suas aplicações seguindo o estilo arquitetural de microserviços.

Os microserviços são uma abordagem de sistemas distribuídos, onde uma única aplicação é desenvolvida como um conjunto de pequenos serviços, cada um executando o seu próprio processo e comunicando numa rede, utilizando mecanismos leves, tais como o protocolo HTTP.

Todos os tipos de aplicações, incluindo as baseadas em microserviços, poderão conter vulnerabilidades, que quando exploradas por um utilizador malicioso poderão causar problemas à empresa que desenvolve a aplicação e/ou aos restantes utilizadores. Uma forma de prevenir tais problemas é através da deteção e correção de vulnerabilidades. No entanto, devido à forma como as aplicações baseadas em microserviços são desenvolvidas, existe uma dificuldade acrescida de como detetar vulnerabilidades resultantes da comunicação entre serviços, uma vez que estas interações são completamente diferentes daquelas que ocorrem dentro de aplicações tradicionais.

Em colaboração com a Checkmarx, desenvolvemos uma solução que deteta vulnerabilidades resultantes da interação entre serviços. Isto foi alcançado com a ajuda do CxSAST, um analisador estático da Checkmarx que deteta vulnerabilidades em código fonte, e das *queries* CxQL, que implementam uma técnica de análise do fluxo de dados. É aplicada uma abordagem composicional, que analisa um serviço de cada vez e de seguida liga os resultados. O nosso trabalho é dirigido a serviços que comunicam através do protocolo HTTP e interagem de uma forma que satisfaz o estilo arquitetural REST.

A nossa solução é composta por um conjunto de *queries* e uma ferramenta externa. As *queries* produzem resultados, procurando indícios de vulnerabilidades no código fonte dos serviços. A ferramenta externa, implementada na linguagem C#, liga estes resultados e define caminhos entre eles. Isto é concretizado através de uma adaptação do algoritmo de pesquisa em profundidade sobre um grafo onde cada vértice é um resultado, ou combinação de resultados produzidos pelas *queries*. De forma a fornecer apenas informação relevante, foram tomadas algumas decisões sobre o tipo de caminhos que pretendemos produzir, tais como a decisão de evitar subcaminhos, que neste contexto são uma fonte de redundância.

Palavras-chave: Análise do fluxo de dados, Análise estática, Grafos, Microserviços, Vulnerabilidades.

Contents

1	Introduction	1
1.1	Context	1
1.2	Goals	2
1.3	Contributions	2
1.4	Place of Internship	3
1.5	Dissertation Outline	3
2	Background	5
2.1	Application Security	5
2.1.1	Basic Concepts	5
2.1.2	Security Vulnerabilities	6
2.1.3	Application Security Testing	9
2.2	Communication	11
2.2.1	Basic Concepts	11
2.2.2	REST	13
2.2.3	HTTP	15
2.3	Other Software Architectural Styles	19
2.3.1	Basic Concepts	19
2.3.2	Monolithic Architecture	23
2.3.3	Service-Oriented Architecture	24
2.3.4	Microservice Architectural Style	24

3	Detecting Vulnerabilities in Microservices	29
3.1	Graph Theory	29
3.1.1	Basic Concepts	29
3.1.2	Representations of Graphs	32
3.1.3	Representations of Attributes	33
3.1.4	Depth-First Search	33
3.2	The Checkmarx SAST Product	37
3.2.1	Control Flow Graph	37
3.2.2	Data-flow Analysis	38
3.2.3	CxSAST	38
3.2.4	CxQL	39
3.3	Microservice Security	40
3.3.1	Security Issues	40
3.3.2	Using CxSAST in Microservice-based Applications	42
3.4	Our Solution	44
3.4.1	Decisions Taken	44
3.4.2	Type of Services Supported	46
3.4.3	Model	49
3.4.4	Queries	60
3.4.5	Results Connector Tool	62
4	Case Studies	75
4.1	Arithmetic Application	75
4.2	Cloud Sec Application	78
4.3	Github Examples	79
4.4	Towards Real-World Applications	80
4.4.1	Externalized Configuration Pattern	80

List of Figures

2.1	Relating tests and result types [25].	11
2.2	The two types of virtualization [60].	20
2.3	Two bounded contexts in an application [72].	22
2.4	An API Gateway and its functions [76].	23
3.1	Various kinds of undirected graphs [83].	32
3.2	Two directed graphs [83].	32
3.3	(a) A directed graph G . (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G [82].	33
3.4	The progress of DFS on a directed graph with timestamps and colors [82].	35
3.5	Two control flow graphs [86].	37
3.6	Interactive interface pointing the presence of an SQL injection [93].	40
3.7	Complete flow F composed of two partial flows from different services.	45
3.8	The interactions within the MVC pattern [95].	46
3.9	A complete flow composed of four partial flows.	50
3.10	The structure of partial flow 2.	51
3.11	The structure of partial flow 4.	52
3.12	A type I flow structure.	53
3.13	A type IV flow structure.	53
3.14	A type IV flow structure composed of a type II and type III flow.	54

3.15	A type V flow structure composed of three type I flows and a type IV flow.	55
3.16	The interactions present in the small example.	56
3.17	The graph of the small example.	56
3.18	A complete flow structure composed of two additional type flows. . .	60
3.19	The first small graph.	66
3.20	The second small graph.	67
3.21	The third small graph.	70
3.22	The fourth small graph.	71
4.1	The database of service Two.	76
4.2	An interaction between the several components present in the Arithmetic application.	77
4.3	A complete flow representing an SQL injection vulnerability.	78
4.4	eShopOnContainers architecture overview [108].	81
4.5	A compose file from the eShopOnContainers application [109].	82
4.6	A draft of a more complete solution.	83

List of Tables

3.1	Relating HTTP request data and action methods.	49
-----	--	----

Chapter 1

Introduction

1.1 Context

Nowadays, software applications are becoming larger, more expensive, and more complex due to high market demands and adoption of new technologies that have emerged in recent years. Several companies, in order to remain competitive and solve problems resulting from traditional architectural styles, have started to develop their applications following the microservice architectural style [1].

Microservices are an approach to distributed systems, where a single application is developed as a collection of small services, each running its own process and communicating over a network using lightweight mechanisms, such as the HTTP protocol. The microservice architecture also integrates new technologies and techniques that have emerged over the last decade, such as on-demand virtualization and infrastructure automation, which helps it avoid some pitfalls of similar architectural implementations. In this approach, services are autonomous, work together, and can be created, initialized, duplicated, and destroyed independently of others [1, 2, 3].

Although the adoption of microservice architectures brings advantages in the development of complex systems, it also presents many challenges. In fact, security is an long-standing problem in network systems, but with microservices it becomes even more challenging. This is due to the large number of entry points and the overload on communication traffic arising from the decomposition of systems into smaller, independent, and distributed software units [1].

But security is not only a serious issue in microservices and, according to the nonprofit foundation Open Web Application Security Project (OWASP), insecure software is undermining critical areas of society such as financial, health, defense, and energy infrastructures. As software becomes increasingly connected and complex,

the difficulty of achieving application security increases exponentially [4].

Given the possibility of security breaches occurring, there is a need to detect and correct potential vulnerabilities in an application after its release, and throughout its software development life cycle. Application security provides measures for this, and aims to protect an organization against external threats and internal risks, ensuring that the code responsible for running an application is secure and free of high risk vulnerabilities [5].

1.2 Goals

Checkmarx is one of the leading companies specializing in application security, offering several products to its costumers.

One of these products is CxSAST, a powerful static analyzer built to discover security vulnerabilities in source code. Without going into much detail, CxSAST normally follows the following process: build a logical graph based on the source code, queries it, and provides the user scan results. Each result provides a brief description of the identified vulnerability, a list of recommendations on how to solve it, an indication of which part of the application is insecure, among other features [6].

With CxSAST in mind and with the intention to increase its coverage, Checkmarx requested a solution to the challenge of detecting vulnerabilities resulting from service-to-service interactions within a microservice-based application. And in this dissertation, the security solution is designed for services that communicate through HTTP and interact in a way that satisfies the REST architectural style.

From the start, it was requested that our solution would have to resort to a compositional approach, i.e., it would have to analyze one service at a time, and then connect the results. It was also established that our implementation would have to find vulnerabilities without hindering the way CxSAST naturally detects vulnerabilities present in a single service.

1.3 Contributions

Our solution aims to define paths in the application through the connection of small results from different services, and thus making it possible to infer the presence of vulnerabilities. This desired solution was realized through the development of new queries, and a new external tool.

In short, when given the source code of a microservice-based application, CxSAST and these new queries will analyze the services, one at a time. In this first part, we search for traces of vulnerabilities, and for each service, we produce a collection of small results. After analyzing all the services and executing all the necessary queries, the external tool connects the small results, producing new and larger results. In the production of these results, we had to be aware of several aspects, such as making proper connections, and not sharing information that has already been shared. If the small results indicate with good certainty the presence of vulnerabilities, and the connections are done correctly, then we can say that the tool accurately detects vulnerabilities resulting from service-to-service interactions.

1.4 Place of Internship

The work presented in this dissertation was partly developed during an internship at Checkmarx in the context of the Master’s Degree in Mathematics and Computing, of University of Minho.

During this internship, I had the possibility to learn several important subjects used in this dissertation. For example,

- In terms of application security, I learned its core concepts and some of the most important security vulnerabilities;
- In the context of software development, I was exposed to some of its techniques, to essential concepts of web applications, and to some of the most important programming languages of today;
- In regards to the comprehension of Checkmarx products, I had access to the fundamental details of both CxSAST and the Checkmarx Query Language.

1.5 Dissertation Outline

The dissertation is organized as follows.

Chapter 2 provides a detailed description of three topics that took a central position in the design of our solution, and can thus be seen as the background of this dissertation. The Section “Application Security” contains a definition of security vulnerabilities, a brief summary of those supported in our solution and a short description of application security testing. Secondly, the Section “Communication” is dedicated to specify the REST architecture and the HTTP protocol. Lastly, the

Section “Other Software Architectural Styles” not only describes microservices, but also briefly defines some of its contrasting architectural approaches.

Chapter 3 describes our solution, and how it came to be. It starts with an overview of graph theory, which was fundamental in producing the new and larger results mentioned in Section 1.3. Afterwards, the necessary information about CxSAST and the Checkmarx query language is provided. After this, we present some specific aspects of security in the context of microservices, and how they relates to the main challenge of this dissertation. Finally, both parts of the solution (the queries and the external tool) are explained in detail.

Chapter 4 covers the case studies and their impact on our solution. Additionally, at the end of this chapter, we present some ideas on how to support an important microservice pattern, which we believe is widely present in real-world applications. We have become aware of its considerable significance towards the end of the testing phase, and so its support can be seen as important future work.

Finally, Chapter 5 contains general conclusions regarding the presented problems and methods.

Chapter 2

Background

In this chapter we take a close look at application security, a type of RESTful communication, and the microservice architecture. Each of the corresponding sections begins with definitions of basic concepts that are needed to explain more advanced ones.

2.1 Application Security

Application security encompasses measures taken to improve the security of an application. Its main objective is to reduce the overall risk that an application poses to an organization. This is achieved through the detection of weaknesses within an application [5, 7].

Before taking a closer look at application security, we need to define some basic concepts.

2.1.1 Basic Concepts

Database

A database is an organized collection of data, generally stored and accessed electronically from a computer system [8].

DBMS

Database Management System (DBMS) is a software system that enables users to define, create, maintain, and control access to the database [8].

Query Language

A query language is a specialized programming language for searching and changing the contents of a database. Even though the term originally refers to a sublanguage for only searching (querying) the contents of a database, modern query languages are general languages for interacting with the DBMS, including statements for defining and changing the database schema, populating the contents of the database, searching the contents of it, and updating its contents [9].

Queries are usually expressed declaratively without side effects using logical conditions. However, modern query languages also provide general programming language capabilities. Most query languages are textual, meaning that the queries are expressed as text string processed by the DBMS [9].

SQL

The most widespread query language is SQL, the standard language used for interacting with relational DBMSs. Queries in SQL are mainly expressed as syntactically sugared relational calculus expressions [9].

Serialization

Serialization is the process of translating a data structure or object state into a format that can be stored, transmitted, and later reconstructed. When a resulting serialization data is reread according to its format, it can be used to create a semantically identical clone of the original object. This process is the opposite operation of serialization, and is called deserialization [10].

Having seen these definitions, we can now take a closer look into application security.

2.1.2 Security Vulnerabilities

As previously mentioned, the main purpose of application security is to identify weaknesses in software that can be exploited by hackers and other external threats. In this context, the word “weakness” can generally be replaced by “security vulnerability”.

A security vulnerability is a weakness in an application, caused either by a design flaw or by an implementation bug. If a vulnerability is discovered by a malicious actor, then the weakness may harm the application that contains it in several different ways, depending on the type of vulnerability and the type of application [11].

There are several online communities looking to improve software security and one of them is OWASP. Every few years, this non-profit foundation launches the OWASP Top 10, a standard awareness document designed to help developers enhance the security of their web applications. It represents a broad consensus about the most critical security vulnerabilities to web applications [12].

According to 2017's OWASP Top 10 [13], the top 10 most critical application security risks are:

A1 - Injection

A2 - Broken Authentication

A3 - Sensitive Data Exposure

A4 - XML External Entities

A5 - Broken Access Control

A6 - Security Misconfiguration

A7 - Cross-Site Scripting

A8 - Insecure Deserialization

A9 - Using Components with Known Vulnerabilities

A10 - Insufficient Logging & Monitoring

The solution presented in this dissertation supports A8 and a variant of A1. We will now give an overview of these two types of vulnerabilities.

SQL Injection

An injection vulnerability allows attackers to relay malicious code through an application to another system. An SQL injection vulnerability, a form of these type of vulnerabilities, allows an attacker to interfere with the statements that an application runs in its database [14, 15].

An SQL injection attack is a code injection technique in which malicious SQL statements are inserted into an entry field in order to effect the execution of predefined SQL commands [16, 17].

A successful SQL injection can read sensitive data from the database, modify database data, execute administration operations on the database, and recover the content of a given file present on the DBMS file system [16].

The following line of code illustrates an example of an SQL statement that might exploit this vulnerability [17]:

```
string statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```

The variable `statement` is the concatenation of three strings:

- "SELECT * FROM users WHERE name = '";
- the value assigned to the variable `username`; and
- "';

In this case, this predefined SQL statement is not written in a way that mitigates potential attacks. The developer who wrote this code intended this SQL statement to be used to extract from the `users` table all the rows from a user with same name as the value from the variable `userName`. To note that in SQL, a single quote marks the start or the end of a string [17].

In this example we assume that the value of the `userName` variable is a user input and that no validation whatsoever occurs in the application in order to verify that this input can lead to a negative outcome. Also in this case, if the user input is crafted in a specific way by a malicious user, this SQL statement may do more than the developer intended. For example, setting the `userName` variable as:

```
' OR '1'='1'
```

renders the following SQL statement:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

If this code were to be used in authentication procedure, then this example would force the selection of every data from all users rather than from one specific user as the developer intended [17]. This is because the evaluation of `'1'='1'` is always true, and thus the comparison predicate:

```
name = '' OR '1'='1'
```

also evaluates to always true. In this case, the SQL statement used is equivalent to:

```
SELECT * FROM users;
```

In order to prevent SQL injection attacks, it is recommended to validate any input that an application receives, and instead of using string concatenation as a way of to create SQL statements, it is advised to use alternative methods provided by trusted libraries [18].

Insecure Deserialization

Insecure deserialization occurs when an application deserializes untrusted data without sufficiently verifying that the resulting data will be valid. Malformed data or unexpected data can be used to abuse the application logic, deny service, or execute arbitrary code, when deserialized [19, 20].

We will now present an example of an exploit of this vulnerability using JSON text to execute arbitrary code. This example was taken from a blog that shows an attack for this vulnerability in C# [21].

For example, in the source code of a web application, if a certain option of a particular JSON serializer is set to a different configuration than the default one, then we have a vulnerability in the code. We will not go into detail about the configuration or the serializer, since it would only make this example more and more complex. So, if a malicious user sends the following JSON text:

```
{"$type": "System.IO.FileInfo, System.IO.FileSystem", "fileName": "web.txt", "IsReadOnly": true}
```

to be deserialized by the serializer class, then it will define the “web.txt” as a read-only file. This file is part of the application and now its content cannot be changed. This of course is only possible if a file called “web.txt” exists and is located in the same folder as the file that performs deserialization. And in this example, the JSON text is written in such a way that the application handles it as C# code.

The example presented shows that the vulnerability in the source code originated from a configuration mistake. This particular attack will probably not create a noticeable negative outcome due to a simple change in the read-only attribute of a file.

In order to prevent this vulnerability, it is recommended to use serialization libraries that only permit primitive data types, or to simply not accept serialized objects from untrusted sources [22].

2.1.3 Application Security Testing

Application Security Testing (AST) products have the purpose of finding vulnerabilities in software applications.

Gartner, a research and advisory company, identifies four main AST technologies [23]:

- Static AST (SAST) technology which analyzes the source, bytecode, or binary code of an application for security vulnerabilities, typically at the programming and/or testing software life cycle phases;

- Dynamic AST (DAST) technology which analyzes applications in their dynamic running state, during testing or operational phases. It simulates attacks against an application, analyzes how the it reacts and, thus, determines whether it is vulnerable;
- Interactive AST technology (IAST) which combines elements of DAST and instrumentation in order to identify vulnerabilities in the application. In this context, instrumentation refers to the use of additional techniques in order to gather information about how the application performs and reacts when it is being tested.
- Software Composition Analysis (SCA) technology which is used to identify open-source and third-party components utilized by the application, their known security vulnerabilities, and typically adversarial license restrictions.

To note that only technologies based on SAST will be important in this work.

Result Types

In SAST-based technologies, a scan is related to four concepts: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) [24].

For each result that a scan produces, a human expert is needed to decide whether or not it represents a vulnerability. If it represents a vulnerability, then we have a true positive. If it does not, then we have a false positive [24].

The part of the application deemed insecure by the result [24]:

- Needs to be fixed in the presence of a TP; or
- Does not need to be fixed in the presence of a FP.

If a SAST tool has not reported a certain security issue at a specific location in the source code, then there are two possible reasons for this [24]:

- The specific location is secure for this vulnerability; or
- The specific location contains the security vulnerability but, due to limitations of the tool, it is not reported.

In the first case we have a true negative, and in the second one we have a false negative.

Figure 2.1 generalizes the ideas presented above, when considering a specific location in the source code and a particular vulnerability. Then:

- The condition is deemed as present when this location contains the vulnerability.
- The condition is deemed as absent when this location does not contain the vulnerability.
- The test is positive when a result is produced.
- The test is negative when a result is not produced.

		condition	
		present	absent
Test	positive	true positive	false positive
	negative	false negative	true negative

Figure 2.1: Relating tests and result types [25].

2.2 Communication

This section describes the type of communication supported by our solution. It is important to understand the concepts behind the REST architecture and the HTTP protocol, as they provides a clearer picture on how services interact with each other. And as such, the following subsection defines some basic concepts related to these two aspects.

2.2.1 Basic Concepts

Network

A network is a large system consisting of many similar parts that are connected together to allow communication between or along the composing parts [26].

Distributed System

A distributed system is a system comprised of components located on a network, which communicate and coordinate their actions by passing messages to one another. Generally, these components interact with each other in order to achieve a common goal [27].

URI

A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource, where the term “resource” is used in a general sense for whatever might be identified by a URI [28].

URL

A Uniform Resource Locator (URL) is a reference to a web resource that specifies its location on a network and a mechanism for retrieving it. URLs are a specific type of URIs, and occur most commonly to reference web pages [29].

Interface

An interface is a shared boundary at which independent and often unrelated components meet and communicate with each other [30, 31].

API

An Application Programming Interface (API) is a computing interface that defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, and so forth [32].

Client–Server Model

Client–server model is a distributed application structure that partitions workloads between service providers and service requesters. In this model, a server is called the provider and a client is called the requester [33].

Client-side and Server-side

Over a network and in a client–server model,

- Client-side refers to operations that are performed by the client [34];
- Server-side refers to operations that are performed by the server [35].

User Interface

A user interface is the means by which a user controls a software application or hardware device [36].

Communication Protocol

A communication protocol is a system of rules that allow two or more entities of a system of communications to transmit information. The protocol defines rules, syntax, semantics, synchronization of communication, and possible error recovery methods [37].

Request/Response Pattern

Request/response is a pattern used in order to communicate through a network, in which one application sends a request message for some data, and another application one receives and processes it, eventually returning a response message [38].

This pattern is typically implemented in a purely synchronous manner. An example of this is when a web service makes a call over HTTP, opens a connection, and waits until the response is delivered or until a timeout period expires. In this case, it can be said that a response message is received promptly and both entities must be available during the time required to complete this request. However, a request–response can also be implemented asynchronously, where instead a response is returned eventually and both entities can be available for the duration of this request [38, 39].

Having defined the basic concepts, we are now ready to have a closer look into the REST architecture and the HTTP protocol.

2.2.2 REST

Representative state transfer (REST) is a software architectural style that defines a collection of constraints to be used when creating web services [40].

There exists six guiding constraints that define a RESTful system. These constraints restrict the ways that the server can process and respond to client requests so that, by operating within these constraints, the system gains desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability. If a system violates any of the required constraints, it cannot be considered RESTful [40].

Architectural Constraints

Although REST is mentioned as an important aspect of this dissertation, in reality not lot of work was put into our solution in relation to it, and thus the following

information is presented as a means to understand better this architectural style. The formal REST constraints are as follows [40, 41]:

- *Client-server architecture*: The principle behind the client-server constraints is the separation of concerns. It looks to separate user interface concerns from data storage concerns.
- *Statelessness*: The client-server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and the session state is held in the client.
- *Cacheability*: Responses must, implicitly or explicitly, define themselves as either cacheable or non-cacheable to prevent clients from providing stale or inappropriate data in response to further requests.
- *Layered system*: The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.
- *Uniform interface*: Uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:
 - Identification of resources: Individual resources are identified in requests. The resources themselves are conceptually separate from the representations that are returned to the client.
 - Manipulation of resources through representations: When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource’s state.
 - Self-descriptive messages: Each message includes enough information to describe how to process the message.
 - Hypermedia as the engine of application state (HATEOAS): Having accessed an initial URI for the REST application, a REST client should then be able to use server-provided links dynamically to discover all the available resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other resources that are currently available.

- *Code on demand* (optional): REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

Resource

The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document, an image, a temporal service, a collection of other resources, a non-virtual object, and so on. REST uses a resource identifier to identify the particular resource involved in an interaction between components [41].

The state of the resource at any particular timestamp is known as resource representation. A representation consists of data, metadata describing the data and hypermedia links which can help the clients in transition to the next desired state [41].

The data format of a representation is known as a media type. The media type identifies a specification that defines how a representation is to be processed. Every addressable unit of information carries an address, either explicitly or implicitly [41].

Resource Methods

Another important aspect of the REST architecture are the resource methods that are used in order to perform a desired transition [41].

Roy Fielding, the creator of REST, has never mentioned any recommendation around which method to be used in which condition. All he emphasizes is that the web service should satisfy the uniform interface constraint [41].

Ideally, everything that is needed to change the resource state shall be part of an API response for that resource. This includes information about the methods and in what state they will leave the representation [41].

There is no official standard for RESTful web APIs because REST is only an architectural style and not a standard in itself. RESTful implementations make use of standards such as HTTP, URI, JSON, and XML [40].

2.2.3 HTTP

The Hypertext Transfer Protocol (HTTP) is a stateless application-level request/response protocol that operates by exchanging messages across a reliable transport [42].

HTTP is a generic interface protocol for information systems. It is designed to hide the details of how a service is implemented by presenting a uniform interface to clients that is independent of the types of resources provided [42].

HTTP is also designed for use as an intermediation protocol for translating communication to and from non-HTTP information systems [42].

HTTP Terminology

The next terms refer to the roles played by participants in, and objects of, the HTTP communication [42, 43].

- Connection: A transport layer virtual circuit established between two programs for the purpose of communication.
- Message: The basic unit of HTTP communication, consisting of a structured sequence of octets transmitted via the connection.
- Request: An HTTP request message.
- Response: An HTTP response message.
- Resource: A network data object or service that can be identified by a URI. Resources may be available in multiple representations (e.g., multiple languages, data formats, size, and resolutions) or vary in other ways.
- Entity: The information transferred as the payload of a request or response. An entity consists of metainformation in the form of entity-header fields and content in the form of an entity-body.
- Representation: An entity included with a response that is subject to content negotiation.
- Client: A program that establishes a connection to a server for the purpose of sending one or more HTTP requests.
- Server: A program that accepts connections in order to service HTTP requests by sending HTTP responses.
- Gateway: A server which acts as an intermediary for some other server.
- Cache: A local store of previous response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cacheable

responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests.

- Cacheable: A response is cacheable if a cache is allowed to store a copy of the response message for use in answering subsequent requests.

The terms “client” and “server” refer only to the roles that these programs perform for a particular connection. The same program might act as a client on some connections and a server on others [42].

Some additional features are as follows [42, 44]:

- HTTP relies upon the URI standard to indicate the target resource and relationships between resources.
- HTTP is defined as a stateless protocol, meaning that each request message can be understood in isolation.
- The start-line and HTTP headers of an HTTP message are collectively known as the head of the request, whereas its payload is known as the body.

HTTP Request Methods

The request method token is the primary source of request semantics. It indicates the purpose for which the client has made this request, and what is expected by the client as a successful result. Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs [45, 46].

The request methods supported in the solution of this dissertation are GET, POST, and PUT. The following is a brief overview of these methods [46, 47]:

- The GET method requests a representation of the specified resource. Requests using GET should only retrieve data;
- The POST method is used to submit an entity to the specified resource, often causing a change in state, or side effects on the server. This method should only be used to create a new representation of a resource; and
- The PUT method replaces all current representations of a target resource with the request payload. This method should generally only be used to update the representation of a resource.

HTTP Requests

An HTTP request consists of the following elements [48]:

- An HTTP method;
- The path of the resource to fetch;
- The version of the HTTP protocol;
- Optional headers that convey additional information for the servers; and
- In some cases, a body for some methods like POST.

HTTP Response

An HTTP response consists of the following elements [48]:

- The version of the HTTP protocol it follows;
- A status code, indicating if the request was successful, or not, and why;
- A status message, i.e., a non-authoritative short description of the status code;
- HTTP headers, like those for requests;
- Optionally, a body containing the fetched resource.

REST and HTTP

In the design of microservices, a popular architectural style for request/response communication is REST. This approach is based on, and tightly coupled to, the HTTP protocol [49].

To better understand the coupling between HTTP and REST, the presence of the following aspects usually define an HTTP-based RESTful API [40]:

- A collection of base URIs, such as “http://api.example.com/collection”.
- A collection of standard HTTP methods (e.g., GET, POST, PUT, and DELETE);
- A collection of media types that define how representations can be processed (e.g., application/json and text/html).

In this context, a REST API endpoint is the location of particular accessible resource [50].

2.3 Other Software Architectural Styles

In a broader sense, a software application can be seen as a program or group of programs designed for end users [51].

The next subsection is dedicated to virtualization and some software development practices, so that we can better understand microservices and the two other architectural styles mentioned in this section.

2.3.1 Basic Concepts

Virtualization

Virtualization is the process of running a virtual instance of a computer system in a layer abstracted from the actual hardware [52].

Operating System

An operating system (OS) is a software system that manages hardware, software resources, and provides common services for computer programs [53].

Operating System Kernel

The OS kernel is a computer program at the core of a OS with complete control over the system. It facilitates interactions between hardware and software components [54].

Virtual Machine

A virtual machine (VM) is a software simulation of a hardware platform that provides a virtual operating environment [55].

Virtualization makes it possible to create multiple VMs, each with their own OS and applications. This is possible through a hypervisor, which is a small software layer that separates VMs and allocates processors, memory, and storage among them [56].

A physical machine on which a hypervisor runs one or more VMs is called a host machine, and each VM is called a guest machine [57].

Container

A container is a virtual runtime environment that runs on top of a single OS kernel and emulates an OS through a container engine. This engine allocates cores and memory to containers, enforces spatial isolation, and provides scalability by enabling the addition of containers [55].

One way to better understand the concept of containers is through its comparison with virtual machines [58]:

- A container virtualizes the operating system. It contains an application, its libraries and dependencies.
- A virtual machine uses a hypervisor to virtualize physical hardware. It contains a guest OS, a virtual copy of the hardware, an application, and its associated libraries and dependencies.

Containers use fewer resources than virtual machines because they share the OS kernel of the machine and do not require an OS per application [59].

A traditional virtual machine and a container implementation are schematically presented in Figure 2.2.

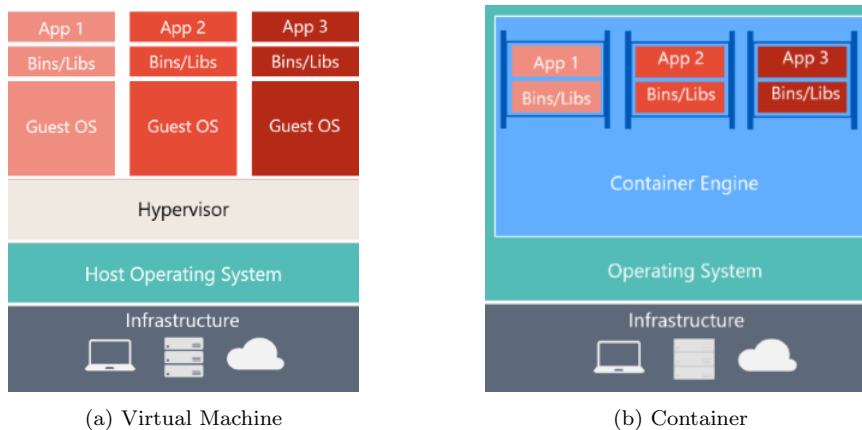


Figure 2.2: The two types of virtualization [60].

Hybrid Container Architecture

A hybrid container architecture is an architecture combining virtualization of virtual machines and containers, i.e., the container engine and associated containers execute on top of a virtual machine [55].

Container Orchestration

Container orchestration is the automated deployment, management, scaling, and networking of containers [61].

Cloud Computing

Cloud computing is the delivery of computing services over the Internet. The use of cloud services lowers operating costs, helps run infrastructure more efficiently, and gives the opportunity of scaling when a service needs it [62].

Loosely Coupled System

A loosely coupled system is one in which each of its components has or makes use of little or no knowledge about the definitions of other separate components [63].

Continuous Integration

Continuous integration (CI) is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration tries to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates [64].

Continuous Delivery

Continuous Delivery (CD) is a software development practice in which software is built in such a way that it can be released to production at any time. It automates the software delivery aspect, and thus is capable of yielding frequent production deployments. An important aspect of CD is that in order to create those production deployments, it is required the presence of manual approval [65, 66, 67].

DevOps

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the development life cycle of an application [68].

Modern day DevOps implements certain practices, such as CI and CD. When both practices are in place, the resulting process is called CI/CD, which includes the full automation on all steps throughout the software development life cycle [69, 70].

Domain-Driven Design

Domain-Driven Design (DDD) is an approach to software development that focus on programming a model that has a rich understanding of the processes and rules of a domain. In this context, a domain is the sphere of knowledge or activity around which the software logic revolves [66, 71].

Bounded Context

Bounded context is a central pattern in Domain-Driven Design. This pattern explicitly defines the context within a model and set boundaries in terms of team organization, in term of usage within specific parts of the application, and in term of physical manifestations such as code bases and database schemas [72, 73].

Domain-Driven Design divides up a large system into bounded contexts, and thus keeps the model strictly consistent within these bounds and not distracted by issues outside of this bounded contexts.

We can define bounded context as a specific responsibility enforced by explicit boundaries [3].

Figure 2.3 illustrates the use of bounded contexts in an abstract application.

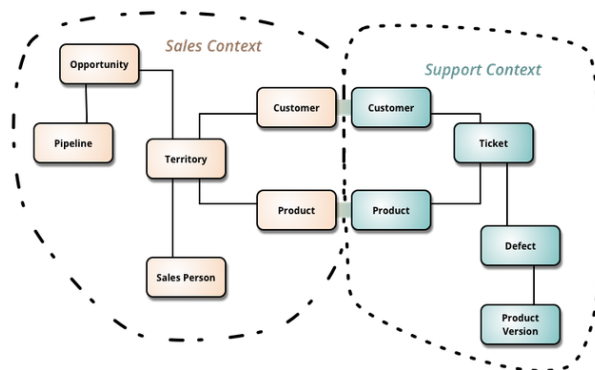


Figure 2.3: Two bounded contexts in an application [72].

System Granularity

System granularity is the extent to which a system is subdivided [74]. In this context, a system can be defined as fine-grained or coarse-grained.

- In a fine-grained approach, a system is subdivided into many small components [74].
- In a coarse-grained approach, a system is subdivided into few large components [74].

API Gateway

In a microservice architecture, an API gateway is a service that sits between the client applications and the services. It is an entry point into the service layer, and is responsible for tasks such as routing requests from client application to services, authentication, SSL termination, and cache [75].

Figure 2.4 depicts an API gateway and shows some of its functions.

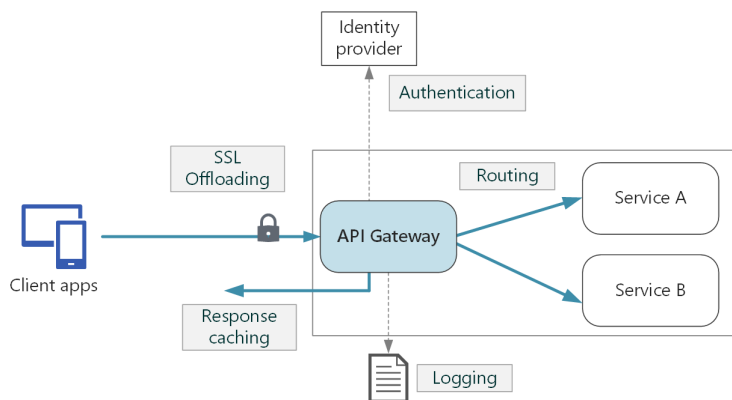


Figure 2.4: An API Gateway and its functions [76].

Now that we saw the necessary definitions, we can now take a closer look into some software architectural styles. The next two architectural styles are usually contrasted with the microservice architectural style.

2.3.2 Monolithic Architecture

A monolithic application is a software application built as a single unit. These type of applications are often put together as three main parts: a client-side user interface, a database, and a server-side application [2].

The server-side application has some important functions like handling HTTP requests and managing data from the database. This side of the application is considered a monolith, that is, a single logical executable. Any changes to the system involve building and deploying a new version of the server-side application [2].

In this architectural style, a change made to a small part of the application requires the entire monolith to be rebuilt and deployed. Over time it is often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module [2].

In the context of scalability, if some parts of a monolithic application require greater resources, then the entire application needs to be scaled. This may be a problem for large applications [2, 3].

2.3.3 Service-Oriented Architecture

Without going into too much detail, Service-Oriented Architecture (SOA) is a distributed systems design approach where multiple services collaborate to provide some end set of capabilities [3, 77].

A service in this context means a realization of some self-contained business functionality. Communication between these services occurs via calls across a network rather than method calls within a process boundary [3].

An important technical concept of SOA is loose coupling. This concept is needed to fulfill the goals of flexibility, scalability, and fault tolerance. In SOA-based applications, loose coupling ensures that a change in one service does not require a change in another [3].

SOA emerged as an approach to combat the challenges of the large monolithic applications. It aims to promote the reusability of software, where two or more end-user applications could both use the same services. It aims to make it easier to maintain or rewrite software, as in principle we can replace one service with another without anyone knowing [3].

An important aspect of SOA is that it can be realized through a variety of technologies and standards [3].

Many of the problems laid at the door of SOA are actually problems with things like communication protocols, the technologies used, a lack of guidance about service granularity, or the wrong approach on picking places to split a system [3].

With all that is needed defined, we can now discuss microservices.

2.3.4 Microservice Architectural Style

Microservices, also known as the microservice architecture, is a fast-moving topic in software development. There exists several sources providing detailed descriptions on what are microservices. We will begin this subsection by mentioning two of the most influential definitions [77]:

- Newman [3] defines microservices as an approach to distributed systems that promote the use of finely grained services build around the following principles: model 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 [2] 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.”

We will now explore and expand relevant parts of both definitions, so we can reach a better understanding of microservices. Also, from now on and to avoid ambiguity, when we say “service” we are referring to the services that constitute a microservice-based application.

We can say that the microservice architectural style was primarily influenced by SOA and the old Unix principle of “do one thing and do it well”. In recent years, microservices have emerged as a trend, or a pattern, from real-world use. This is evident because they use practices that preceded them, such as domain-driven design, continuous delivery, on-demand virtualization, and infrastructure automation [3, 77].

Comparison with SOA

Regarding the comparison with SOA, the majority of opinions fall into one of the two following categories: microservices are SOA, or microservices are a separate architectural style [77].

For example, Newman [3] sees microservices as one way of doing SOA right and Lewis and Fowler [2] both see microservices as a new architectural style that can be contrasted against SOA.

In this work, we take the stance that the microservice architectural style is an implementation of SOA, and hence that it does not constitute a new architectural style. As previously mentioned, many problems with SOA are actually problems involving technologies and paradigms used in their development. And since some new technologies and paradigms have emerged or evolved in the last few years, then the adoption of microservices do not suffer much of the pitfalls of SOA.

In short, we agree with the conclusion taken by Zimmermann [78] in his literature review: “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)”.

Service

Lewis and Fowler [2] define a service as an out-of-process component, that communicates with a mechanism such as a web service request, or remote procedure call. Component in this definition means a unit of software that is independently replaceable and upgradeable.

It is common for a collection of small autonomous teams to be responsible for the development and deployment of one or more related services. Thus, each team can develop, deploy, and scale its services independently from all other teams [79].

Communication

All the communication between services is via network calls, to enforce separation between the services and avoid the perils of tight coupling [3].

The two protocols most commonly used are HTTP request/response with resource APIs and lightweight messaging [2].

The first approach includes the already mentioned communication through an HTTP-based RESTful API.

The second approach utilizes messaging over a lightweight message bus. This enables asynchronous communications between services so that the sending service does not need to wait for the reply of the receiving service [2, 80].

Deployment

Microservices can either be deployed in virtual machines or lightweight containers. The use of containers for deploying microservices is preferred due to their simplicity, lower cost, and their fast initialization and execution [1].

Lightweight containers remove the dependencies of the underlying infrastructure, which reduces the complexity of dealing with different platforms. Consequently, this allows the microservice architecture to test and deploy single services in separate containers over a network. Lastly, containers can provide several other advantages, such as smoother CI/CD [81].

Resilience

Microservice applications need to be designed in a way that can tolerate the failure

of services. If a component fails, but that failure does not cascade, then it should be possible to isolate the problem and enable the rest of the system to carry on working [2, 3].

Since services can fail at any time, it is important to be able to detect the failures quickly and, if possible, automatically restore the service. These applications put a lot of emphasis on real-time monitoring, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received) [2].

It is important to ensure that microservice systems can properly embrace this improved resilience, because there exists new sources of failure that come from the adoption of distributed systems, as networks can and will fail [3].

Decentralized Data Management

Decentralization of data management means that conceptual models will differ between systems. A useful way of thinking about this is with bounded context. Domain-driven design divides a complex domain up into multiple bounded contexts and maps out the relationships between them [2].

Microservices also decentralize data storage decisions. Each service manages its own database, either by using different instances of the same database technology or totally different database systems [2].

Decentralizing responsibility for data across microservices has implications for managing updates. The common approach to dealing with updates has been to use transactions to guarantee consistency when updating multiple resources [2].

Scalability

With the use of smaller services, we can just scale the services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware [3].

Loose Coupling

An important feature in microservices is loose coupling. When services are loosely coupled, a change to one service should not require a change in another. In this architectural style, loose coupling means that each component has less dependencies on other separate components, which makes the deployment and development of microservices more independent [3, 81].

Loose coupling allows testing in isolation from the rest of the underlying systems and enables development using different programming languages and database technologies [1].

Fine Granularity

A microservice architecture should be fine-grained, which means that there must be a minimum of centralized service management [81].

Advantages and Disadvantages

Microservices are highly modular, distributed systems that can be reusable through a network-exposed API. This implies that microservices inherit advantages and disadvantages of both distributed systems and web services [77].

The most frequently listed benefits of the microservice architecture style are organizational alignment, faster and more frequent releases of software, independent scaling of components, and overall faster technology adoption [77].

The disadvantages of microservices include the need to make multiple design choices, the difficulty of testing and monitoring, the design for failure, operational overhead when compared to typical non-distributed solutions, and the appearance of new security challenges [2, 77].

The last mentioned disadvantage shows the relevance in the request made by Checkmarx. The next chapter gives more information about this topic.

Chapter 3

Detecting Vulnerabilities in Microservices

This chapter introduces our solution, whose goal is to detect vulnerabilities resulting from service interactions. This chapter also describes some parts of graph theory, CxSAST, and microservice security.

3.1 Graph Theory

Some concepts presented in this section will be of importance when introducing the CxSAST tool in Subsection 3.2.3. Moreover, some ideas, concepts, and results of this theory were instrumental in the development of our external tool, presented in Subsection 3.4.5.

The concepts and results that follow came from the book Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein [82].

3.1.1 Basic Concepts

A *graph* G is a pair (V, E) , where:

- V and E are finite sets.
- The set V is called the *vertex set* of G , and its elements are called vertices.
- The set E is called the *edge set* of G , and its elements are called edges.

There exists two important types of graphs:

- An *undirected graph* $G = (V, E)$ is a graph where the edge set $E \subseteq \{\{x, y\} \mid x, y \in V \text{ and } x \neq y\}$; and
- A *directed graph* $G = (V, E)$ is a graph where the edge set $E \subseteq \{(x, y) \mid x, y \in V\}$, that is, $E \subseteq V \times V$.

By convention, in undirected graphs we will use the notation (u, v) to denote an edge, and we will consider (u, v) and (v, u) to be the same edge. It is important to note that although both types of graphs are of relevance to the rest of this section, only the directed ones will be of importance beyond this section.

Incidence

If (u, v) is an edge in a directed graph,

- We say the (u, v) is *incident from* or *leaves* vertex u ; and
- We say the (u, v) is *incident to* or *enters* vertex v .

Adjacent Vertex

- If (u, v) is an edge in a graph, we say that vertex v is *adjacent* to vertex u . When the graph is undirected, the adjacency relation is symmetric.

Degree

In a directed graph,

- The *out-degree* of a vertex is the number of edges leaving it;
- The *in-degree* of a vertex is the number of edges entering it; and
- The *degree* of a vertex is its in-degree plus its out-degree.

Path

- A *path* of length k from a vertex u to vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$. Note that the *length* of a path corresponds to the number of edges in the path.
- A path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ is said to *contain* the vertices $v_0, v_1, v_2, \dots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$.
- If there is a path p from u to u' , then u' is *reachable* from u via p .

Simple Path

- A path is *simple* if all vertices in the path are distinct.

Subpath

- A *subpath* of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is a contiguous subsequence of its vertices. That is, for any $0 \leq i \leq j \leq k$, the subsequence of vertices $\langle v_i, v_{i+1}, \dots, v_j \rangle$ is a subpath of p .

Cycle

- In a directed graph, a path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a *cycle* if $v_0 = v_k$ and the path contains at least one edge. A *self-loop* is a cycle of length 1.
- In an undirected graph, a path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a cycle if $k \geq 3$ and $v_0 = v_k$.
- A graph with no cycles is called *acyclic*.

Connected and Strongly Connected

- A directed graph is *strongly connected* if every two vertices are reachable from each other.
- An undirected graph is *connected* if every vertex is reachable from all other vertices.

Tree

- A *tree* is a connected and acyclic undirected graph.

Forest

- A *forest* is an acyclic undirected graph.

Subgraph

- A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- Given $V' \subseteq V$, the *subgraph of G induced by V'* is the graph $G' = (V', E')$, where

$$E' = \{(u, v) \in E \mid u, v \in V'\}$$

Figures 3.1 and 3.2 illustrates some of these definitions.

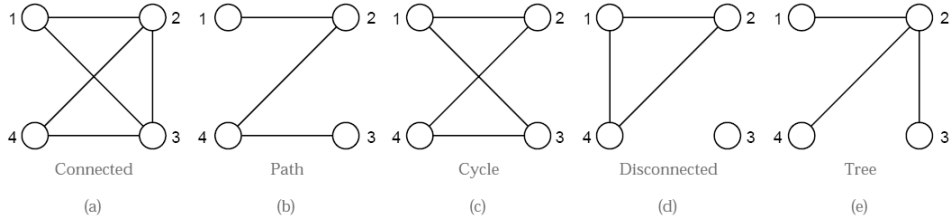


Figure 3.1: Various kinds of undirected graphs [83].

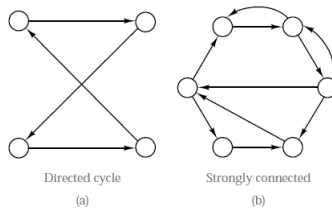


Figure 3.2: Two directed graphs [83].

3.1.2 Representations of Graphs

The two most common computational representations of graphs are as *adjacency lists* and as *adjacency matrices*. Either of these representations applies to both directed and undirected graphs.

The adjacency-list representation provides a compact way to represent sparse graphs, i.e., those for which $|E|$ is much less than $|V^2|$.

The adjacency-matrix representation, however, is preferable when the graph is dense, i.e., $|E|$ is close to $|V^2|$.

Adjacency-List Representation

The adjacency-list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex of V .

For each $u \in V$, the adjacency-list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G .

Adjacency-Matrix Representation

For the adjacency-matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \dots, |V|$ (or $0, 1, \dots, |V| - 1$, depending on the coding

language) in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Figure 3.3 depicts the two representations of a particular directed graph.

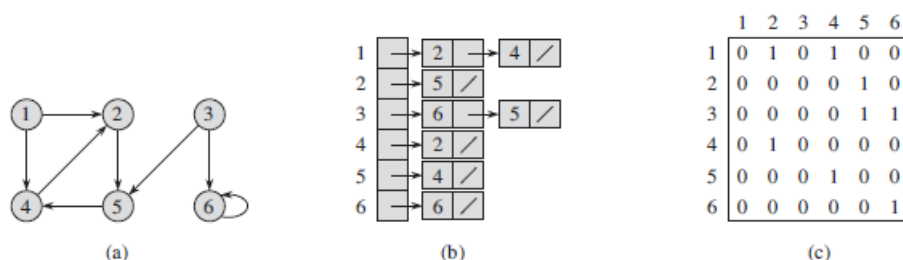


Figure 3.3: (a) A directed graph G . (b) An adjacency-list representation of G . (c) The adjacency-matrix representation of G [82].

3.1.3 Representations of Attributes

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. For example, we use $v.d$ to denote a attribute d of v .

For adjacency lists, one way to represent vertex attributes is to use additional arrays. For the attribute d , there could exist an array $d[1 \dots |V|]$ that parallels the Adj array.

3.1.4 Depth-First Search

Given a graph $G = (V, E)$ and a distinguished source vertex s , the depth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s . This algorithm outputs a forest comprised of several trees. The input graph $G = (V, E)$ may be directed or undirected and in what follows the graphs are represented using the adjacency-list representation.

The strategy followed by depth-first search is, as its name implies, to search “deeper” in the graph whenever possible.

Depth-first search (DFS) explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of edges of v have been explored, the search “backtracks” to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that

are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

Depth-first search may construct several trees. Whenever the search discovers a vertex v during a scan of the adjacency list of an already discovered vertex u , the vertex v and the edge (u, v) are added to a tree. We say that u is the predecessor or parent of v in a depth-first tree.

To keep track of progress, depth-first search colors each vertex white, gray, or black. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint. So if the graph being traversed contains cycles, this algorithm does not reach the same vertex a second time and thus prevents infinite recursion. If a grey colored vertex is reached, that means the graph has a loop.

This algorithm attaches two additional attributes to each vertex in the graph. We store the color of each vertex $u \in V$ in the attribute $u.color$ and the predecessor of u in the attribute $u.\pi$. If u has no predecessor, then $u.\pi = \text{NIL}$.

Therefore, the predecessor subgraph of a depth-first search is denoted as $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(v.\pi, v) \mid v \in V \text{ and } v.\pi \neq \text{NIL}\}.$$

The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. The edges in E_π are called tree edges.

Figure 3.4 shows the order of visit in a graph under the depth-first search (from (a) to (p)). It shows the progress of DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Timestamps within vertices indicate discovery time and finishing times. Non tree edges are labeled B, F, or C according to whether they are back, forward, or cross edges. The idea behind these labels is as follows:

- Back edges are edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered back edges.
- Forward edges are edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- Cross edges are all other non tree edges.

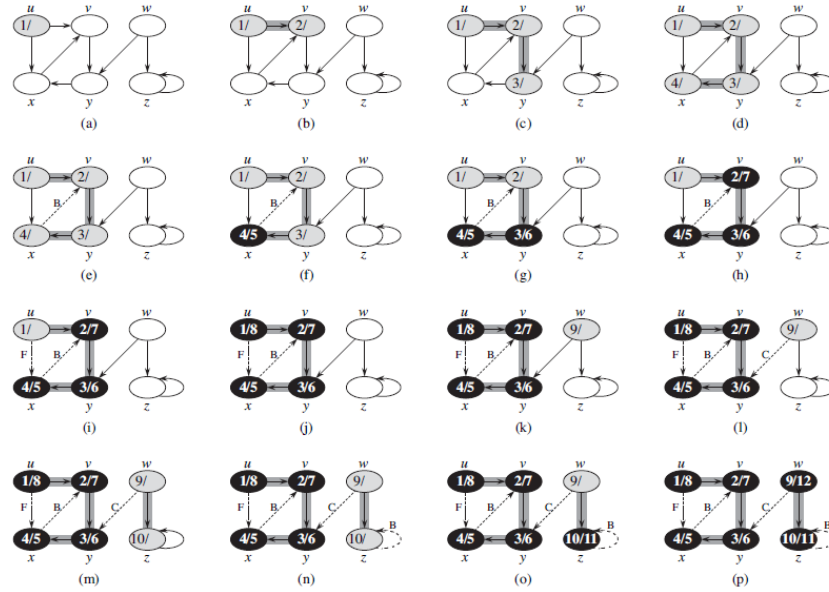


Figure 3.4: The progress of DFS on a directed graph with timestamps and colors [82].

The following pseudocode is the basic depth-first search algorithm.

Algorithm 1 Depth-first search

```

1: procedure DFS( $G$ )
2:   for each vertex  $u \in V$  do
3:      $u.color := \text{WHITE}$ 
4:      $u.\pi := \text{NIL}$ 
5:   end for
6:   for each vertex  $u \in V$  do
7:     if ( $u.color = \text{WHITE}$ ) then
8:       DFS-VISIT( $G, u$ )
9:     end if
10:  end for
11: end procedure

```

Algorithm 2 Depth-first search visit

```

1: procedure DFS-VISIT( $G, u$ )
2:    $u.color := \text{GRAY}$ 
3:   for each vertex  $v \in G.Adj[u]$  do
4:     if ( $v.color = \text{WHITE}$ ) then
5:        $v.\pi := u$ 
6:       DFS-VISIT( $G, v$ )
7:     end if
8:   end for
9:    $u.color := \text{BLACK}$ 
10: end procedure

```

Procedure DFS works as follows:

- Lines 2–5 paint all vertices white and initialize their π attributes to NIL;
- Lines 6–10 check each vertex in V in turn and, when a white vertex is found, visit it using DFS-VISIT;
- Every time DFS-VISIT(G, u) is called in line 8, vertex u becomes the root of a new tree in the depth-first forest.

In procedure DFS-VISIT.

- For each call DFS-VISIT(G, u), vertex u is initially white;
- Line 2 paints u gray;
- Lines 3–8 examine each vertex v adjacent to u and recursively visit v if it is white;
- As each vertex $v \in Adj[u]$ is considered in line 3, we say that edge (u, v) is explored by the depth-first search;
- Finally, after every edge leaving u has been explored, line 9 paints u black.

Before we describe the time complexity of DFS, we need to introduce the big-theta notation. Given two numeric functions f and g , we denote $f(n) = \Theta(g(n))$ when [84]:

$$\exists c_1, c_2, n_0 \in \mathbb{N}, \forall n > n_0 . c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Also, in order to present the time complexity of DFS, we utilize aggregate analysis. This type of analysis considers both the costly and less costly operations throughout the whole series of operations in an algorithm. In short, this analysis considers the worst-case run time per operation, rather than per algorithm [85].

The time complexity of the depth-first search algorithm shown next is from the book Introduction to Algorithms [82] mentioned at the start of this section.

Time Complexity

In the procedure DFG, the loops on lines 2–5 and lines 6–10 take time $\Theta(|V|)$, exclusive of the time to execute the calls to DFS-VISIT.

The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint it gray. During an execution of DFS-VISIT(G, v), the loop on lines 3–8 executes $|Adj[v]|$ times. Since

$$\sum_{v \in V} |Adj[v]| = \Theta(|E|),$$

the total cost of executing lines 3–8 of DFS-VISIT is $\Theta(|E|)$, that is, it takes $\Theta(|E|)$ to check all elements of the adjacency-list representation. The running time of DFS is therefore $\Theta(|V| + |E|)$.

3.2 The Checkmarx SAST Product

The beginning of this section acts as a bridge between graph theory and CxSAST. The notions of control flow graph and data-flow analysis are essential in order to understand CxSAST and our solution.

3.2.1 Control Flow Graph

A control flow graph (CFG) is a representation of all paths that might be traversed through a program during its execution. A control flow graph is a directed graph in which the vertices represent basic blocks and the edges represent control flow paths [86, 87].

A basic block is a linear sequence of program instructions having one entry point and one exit point. An entry point is the first instruction executed and an exit point is last instruction executed [87].

It is important to define two special types of basic blocks: entry and exit blocks. An entry block is where a control flow enters a graph and an exit block is where all control flows leave [86].

A basic block may be preceded or succeeded by many other basic blocks. In a program, entry blocks might not have predecessors and exit blocks will never have successors [87].

Figure 3.5 illustrates two control flow graphs. The one on the left represents a simple if-then-else statement, and the second one represents a simple while loop [86].

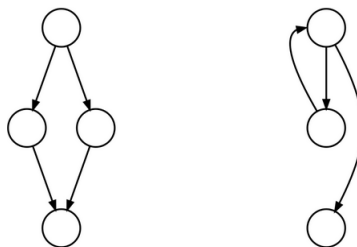


Figure 3.5: Two control flow graphs [86].

3.2.2 Data-flow Analysis

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A CFG of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate [88].

Data-flow analysis is used to collect run-time information about data in software while it is in a static state. It is important to note that data flow analysis may construct a data-flow graph (DFG), i.e., a graph representing the flow of data between operations [88, 89, 90].

3.2.3 CxSAST

The work done in this dissertation is in the field of SAST-based technologies, and in this case, on Checkmarx SAST (CxSAST).

Without needing to build or compile the source code of a software project, CxSAST builds a data-flow graph of the code using data flow analysis. CxSAST can then data-mine any aspect of this data-flow graph with queries written in the patented Checkmarx Query Language (CxQL). These queries produce a list of results, where each result can be a single code element or a flow [6]:

- *Code elements* are vertices in the data-flow graph. They can be variables, method invocations, assignments, etc.
- A *flow* is a particular path in the data-flow graph representing a possible data change in the code.

To better understand this topic of data flow analysis in CxSAST, suppose that exists two different code elements u, v of a DFG produced in a CxSAST scan. Then, we say that v is *data influenced by* u when v is reachable from u .

For example, in the C# code shown below, variable d is data influenced by variables a and b , but it is not data influenced by variable c .

```
int a = 5;
int b = 6;
int c = 7;
int d = a + b;
```

CxSAST provides scan results either as static reports, or in an interactive interface that enables tracking runtime behavior per vulnerability through the code, and provides tools and guidelines for remediation [6].

In the domain of CxQL and CxSAST, a complementary tool to highlight is Cx-Audit. This tool complements CxSAST by enabling the creation and customization of queries [91].

3.2.4 CxQL

Most of the queries equipped for CxSAST follow the idea behind *taint analysis*. That is, these queries attempt to identify variables that have been “tainted” with user controllable input and trace them to possible vulnerable functions. To simplify, we denote user controllable input as *source elements* and possible vulnerable functions as *sink elements* [89].

In other words, queries check where the source code allows information to pass from a source element to a sink element without being properly sanitized. This is done primarily by searching for source elements, sink elements and sanitizing elements, as highlighted in the following example relative to an SQL injection query. A sanitizing element is a special code element that can claim that if a flow from a sink element to source element passes through it, then that flow is not “tainted” [92].

Thus, most queries have the following structure, or something similar to it [92]:

1. Find a certain type of input in the source code;
2. Find a certain type of effect on database, operating system, or other output in the source code;
3. Find places where the source code sanitizes input;
4. Define paths from input source (1) to sink (2) that do not pass through sanitation (3)

For example, the Java SQL injection query is the following [92]:

```
CxList inputs = Find_Interactive_Inputs();
CxList db = Find_DB();
CxList sanitized = Find_Sanitize();
result = All.FindSQLInjections(inputs, db, sanitized);
```

Each of the above four lines calls a function and these functions are actually other queries [92].

Many of these building-block queries are used by multiple higher-level queries. So, for example, if it is added a custom code element to a building-block query that performs sanitation, then that code element will be recognized as sanitized by all queries that use this building-block query.

Figure 3.6 illustrates an example of a scan result showing an SQL injection vulnerability.

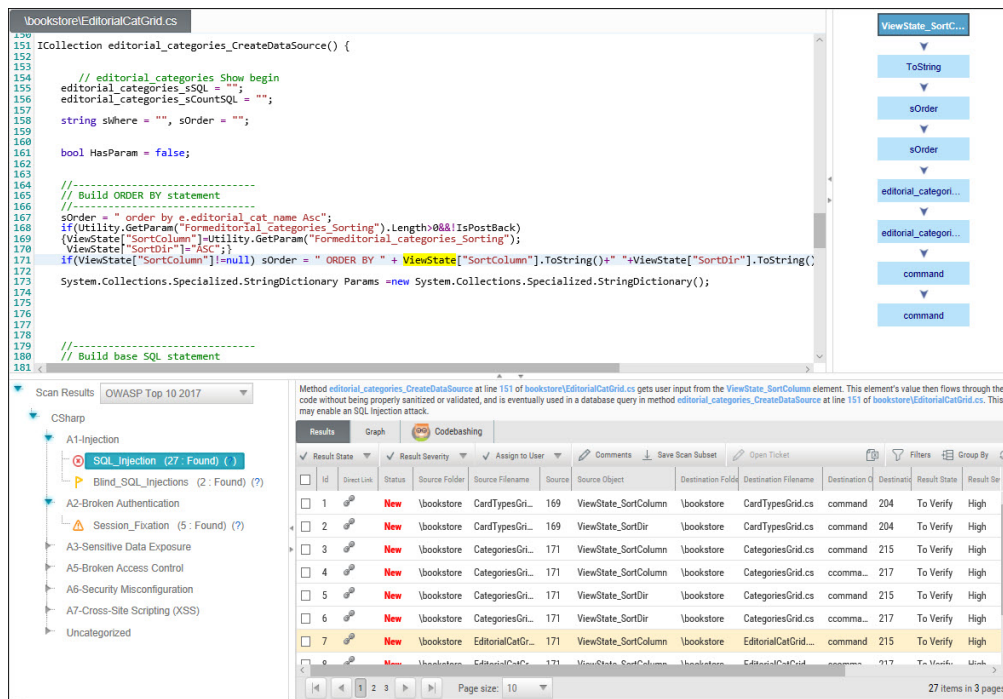


Figure 3.6: Interactive interface pointing the presence of an SQL injection [93].

3.3 Microservice Security

The microservice architectural style brings with it new security threats and vulnerabilities that were not present in traditional monolithic applications. And thus, microservice security can be seen as a multifaceted problem that depends on several aspects related to what defines an application of this type, as well as the underlying technologies used [1, 77].

3.3.1 Security Issues

In order to give a general idea about microservice security, this subsection will address some security issues resulting from service interactions.

Although this subsection concentrates on the new challenges posed by microservice security, our solution for vulnerability detection in microservice-based applications does not directly help CxSAST support all the security issues highlighted here.

Network issues

Network security is a critical aspect in a microservice architecture, since there will be frequent communications among services, and if a network is not secure, serious problems may occur. For example, a network attack that causes a delay or a node outage could paralyze an entire application [81].

The attack surface is much larger compared to traditional applications because services interact with each other through APIs exposed to a network, and these APIs are independent of machine architectures and even programming languages [94].

In the domain of network security, traditional typical threats mainly include Man in the Middle (MITM) and Denial of Service (DoS) [81].

In short, a MITM attack can intercept normal network communication data and manipulate it without both sides of the communication being aware of the attack [81].

A DoS attack is a network attack method that is normally used to disable servers or networks. This attack intentionally exploits network protocol flaws or depletes the resources of the object with the aim of causing the target servers or network to stop responding to users or even to simply collapse them [81].

Permission Issues

Trust is an important dimension in microservice security. Microservice architecture should verify the authenticity of every service, because if a single service is controlled by an attacker, the service can maliciously influence other services. Granting permissions to a user beyond the scope of the necessary rights may allow that user to obtain or change information in improper ways. Therefore, careful delegation of access rights can limit attackers from damaging a system [81].

Authentication and authorization are core concepts when it comes to anything interacting with the microservice system. In the context of security, authentication is the process by which it is confirmed that a party is who they say they are. Generally, when we are talking abstractly about who or what is being authenticated, it is referred to that party as the principal [3].

Authorization is the mechanism of specifying what the principal is allowed to do. Once a principal has been verified through the authentication process, authorization determines what permissions they have [3].

Data issues

Services need to use complex types of communication due to their fined-granularity. Not only is there a risk that message data can be intercepted, but there is also a threat that competitors may be able to infer business operations from message data. Since services are often deployed in cloud environments, they also suffer from privacy issues where cloud consumers may have their stored information compromised or used inappropriately [81].

To ensure data confidentiality, integrity, and availability, the microservice provider must give minimum data security capabilities, including an encryption schema to protect all data in the shared storage environment, strict access controls to prevent unauthorized entree, and safe storage for scheduled backup data [81].

Container issues

Since the existence of containers is highly relevant to the development of microservices, container security is also very important. To make the development and implementation of services more agile, developers can use containers in the cloud. If this happens, using the same kernel as the host of different containers can make it possible for attackers to gain unauthorized access to a container when users are unaware [81].

Container isolation is a critical point in container security. Some containers may use different techniques to isolate resources such as users, processes, networks, and devices [81].

Network isolation is necessary in containers. For example, with the absence of network isolation, if two different containers want to run the same web application on the same port from the same host, then there will be a conflict [81].

In the case of resources such as CPU and memory, if an attacker controls the amount of resources that any container can use, there may be a DoS attack and other container resources may be depleted [81].

3.3.2 Using CxSAST in Microservice-based Applications

According to Gartner, microservice security is beginning to appear as an important trend in application security testing. Gartner also underlines that it saw a 60% increase in the number of clients asking about container security and assumes that by 2025, 70% of attacks against containers will be from known vulnerabilities and misconfigurations that could have been remediated [23].

AST-based technologies that attempt to fully cover microservice security must be attentive to various types of issues, such as those highlighted in the previous subsection and others not mentioned, such as cloud and orchestration issues. In addition, application-level security problems common in traditional applications, such as SQL injection and insecure deserialization, still need to be supported. Although not mentioned in the previous section, it remains equally important to prevent the exploitation of these vulnerabilities in microservice-based applications.

Limitations of CxSAST

When scanning microservice-based applications, CxSAST by itself has no means of detecting vulnerabilities that result from service-to-service interactions. This is attributed to the way services communicate, where CxSAST does not recognize this type of connections made over a network. And yet, some information about these interactions is present in the source code of the application. This problem is evident when an application that does not have a single isolated service is scanned. In this example and in an ideal world, CxSAST is expected to produce a single DFG, but in reality it creates a set of DFGs, one for each service. Therefore, CxSAST cannot produce flows between services.

Benefits of a Compositional Approach

Checkmarx stipulated from the beginning that a solution for vulnerability detection in microservice-based applications would have to use a compositional approach. One of the main reasons why this is required is because of how microservices are developed. We know that ideally a collection of teams is responsible for the development of one or more services, and that this type of applications adhere to loose coupling. This implies that each service is developed at different rates.

When a new client requests the complete analysis of an application, our solution will analyze all services and save part of the scan data for future analysis. The next scan uses the data from the previous one, and allows the problem of checking the security of the application to be satisfied by the scans in services that have changes. The process of saving data for each scan and the use of a compositional approach implies that it is not necessary to analyze the entire application after the first scan.

The next section describes our solution in much more detail.

3.4 Our Solution

This section starts with the disclosure of the decisions taken regarding the development of our solution. Afterwards, it is identified the type of microservice-based applications supported in this work, and the model that specifies how the new queries and the new external tool were to be developed. Finally, at the end of this section, we describe in detail our solution for vulnerability detection.

3.4.1 Decisions Taken

As already mentioned, our solution does not directly help CxSAST to support all the security issues highlighted in Subsection 3.3.1. This responsibility is assigned to the various security experts of Checkmarx.

The main objective of this solution is to produce paths between services. In the context of data-flow analysis, there is some evidence in the source code of a service that certain parts of it can influence other services. And in the case of communication through HTTP-based APIs, this evidence may very well consist of URLs and HTTP methods.

Suppose there exists an application that uses an HTTP-based RESTful API in the communication between abstract services A and B. Suppose it also exists a particular SQL injection flow, called F, from service A to service B. Flow F could ideally be broken down into two other flows:

- Flow F1, a flow from a user input to a particular service exit point, that is, from a user input to an HTTP request method; and
- Flow F2, a flow from a service entry point to a method that processes an SQL command.

In the above example, flow F1 represents only source code of service A, flow F2 represents only source code of service B, and flow F represents source code of both services (a service entry point is later defined in Subsection 3.4.2).

From now on we will refer to flows like F1 and F2 as partial flows and flows like F as complete flows. A partial flow is one that represents a path contained in only one service. A complete flow is one that represents a path that goes through at least two services. We will also refer to complete flows produced by our external tool as complete results, because this tool only outputs a subset of all complete flows.

Figure 3.7 illustrates a complete flow composed of these two partial flows:

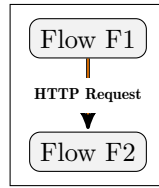


Figure 3.7: Complete flow F composed of two partial flows from different services.

From the start, Checkmarx made it clear that no changes had to be made in the source code of CxSAST, and the solution to be developed should scan microservice-based applications in a compositional way. This means that each service should be scanned separately from the others, producing in the end a set of data structures that represent a single loosely coupled application. As there are certainly several ways to tackle the problem at hand, some design decisions had to be made. The main decisions are presented next.

First, it was planned that our solution could make connections between all the produced DFGs. But considering the time available for this dissertation, we decided to reject this idea, as it might well have proven to be too complex.

The idea considered afterwards was to connect scan results. In the context of a scanning procedure, scan results are produced at a later stage than the set of DFGs. Ultimately, this was the chosen approach because of its simplicity, and the fact that it could benefit from some of the knowledge acquired during the internship with Checkmarx, namely knowledge about web development and the Checkmarx query language.

This approach requires the creation of new queries in order to produce partial flows. It should be noted that each flow produced will not be able to declare the presence of vulnerabilities on its own, since its sole purpose is to be connected with others so that complete flows can be obtained.

Complete results can be categorized as true positives or false positives. And this by itself brings the need to reduce the number of false positives, as we primarily seek to avoid them. To do this, we developed a model responsible for deciding what the structure of the partial flows could be, how to connect them, which partial flows should be filtered out, etc.

CxSAST makes it possible to treat each service as if it were a single application in order to produce partial flows. Using an external tool and following a model, we can later connect these flows. Thus, in this process we apply the idea of compositionality. Before presenting the approaches used in our solution, let us see what type of services are supported.

3.4.2 Type of Services Supported

It was previously mentioned that the goal of this dissertation is to support services that communicate through HTTP-based RESTful APIs. Something that has not been explicitly stated is that developers have several ways of writing their applications in order to implement this type of communication.

An important restriction of our solution is that services must have their own HTTP-based RESTful API. Also, since it is very likely that the whole lifecycle of a service is owned by a small team, we have to think from the point of view of a single service. Hence we can think the communication of a service as if it was the interaction between itself and its clients. The clients of a singular service can be other services, a web application that handles user input, and even other applications developed by other organizations than the one we are looking at [79].

In this work, we restrict to only one way a service can receive data from its clients, that is, there exists only one type of service entry point for all manner of clients. This idea is closely related with the model–view–controller pattern, detailed below.

MVC Pattern

Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divides the logic of a program into three interconnected components [95]:

- The model - the central component of the pattern. It directly manages the data, logic and rules of an application.
- The view - any representation of information such as a chart, diagram or table.
- The controller - receives the input, optionally validates it and then passes it to the model.

Figure 3.8 illustrates the interactions within the MVC pattern.

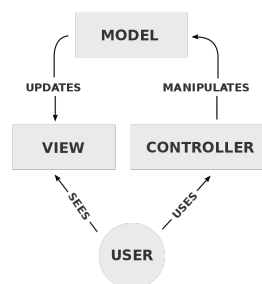


Figure 3.8: The interactions within the MVC pattern [95].

In order to recognize a flow that represents a communication between various services, all but the last service of the communication will have to:

- Use a collection of controller files that implement REST API endpoints; and
- Execute at least one HTTP request using a hard coded URL, that is, the value of the address used in the HTTP request must be directly embedded in the source code of the service.

Whereas the last service only needs to satisfy the first condition.

It is important to note that in order to produce complete flows, HTTP requests must use URLs directly embedded into the source code of the service that initiates the request.

In the interaction between two services, the first service that starts the interaction will have to satisfy both conditions, and the other service will have to satisfy the first one.

The controllers mentioned here are very similar to the MVC ones. But in this case we are talking in the context of Web APIs, where controllers are simply components that handle HTTP requests [96].

In this work, controllers are the recognized service entry points. Therefore, at least one controller file must be present in an interaction between two services.

A controller class generally consists of a set of request handler methods. Each method implements a REST API endpoint and the method parameters represent values from an HTTP request [79].

Our solution will only recognize service communications written in the C# and Java programming languages. It is possible to recognize a simple interaction of a service written in C# and another written in Java, since the product of this work follows an essentially compositional pattern. In theory, this pattern also allows the recognition of communications written in other languages. It is important to emphasize that most of the work done in this dissertation is directed at services written in C#, and that the support for those in Java is basic in nature.

From now on, only C# directed ideas will be presented, since the Java version is very similar and follows the same ideas. All the functionalities supported in the Java side are supported in the C# side.

The next subsection is closely related to C# and references the ASP.NET Web API framework. This was the supported framework and will give an idea about controller routing. This concepts are also available in Java through other means, such as the Play Framework [97].

The following information comes from the C# Microsoft documentation [96].

Controller Routing

In the ASP.NET Web API framework, a controller class is a class that handles HTTP requests, where its public methods are called action methods. When receiving an HTTP request, this framework tries to route it to a particular action method using a routing table. If no route matches, the client receives a 404 HTTP error message.

Each entry in a routing table contains a route template. When receiving an HTTP request, the Web API framework tries to match the URI with one of the existing route templates. For example, the following URIs match the route template “api/<controller>/[<id>]”:

- api/contacts;
- api/contacts/1;
- api/products/0.

However, the following URI does not match, since it lacks the “api” segment:

- contacts/1.

In this example, the route template has the “api” part as a literal path segment, and “<controller>” and “[<id>]” as placeholder variables. Also, the “[<id>]” segment is optional.

Afterwards, once a matching route is found, the framework selects the controller and the action method, taking into account:

- The value of the “<controller>” variable, in order to find the controller.
- The HTTP verb used in the request, and the action methods associated with this verb, in order to find the correct action method.
- Other placeholder variables in the route template, such as “[<id>]” shown before.

Table 3.1 shows possible HTTP requests, along with the action method that gets invoked for each.

HTTP Request Method	Request URI	Action	Parameter
GET	api/contacts	GetAllContacts	(none)
GET	api/products	GetAllProducts	(none)
GET	api/products/1	GetProductById	1
POST	api/products/2	CreateProduct	2
POST	api/products/3	CreateProduct	3
PUT	api/products/4	UpdateProduct	4
PUT	api/contacts	(no match)	

Table 3.1: Relating HTTP request data and action methods.

It is important to highlight the following notes about this table:

- All requests match the routing template “api/<controller>/[<id>]”.
- Both POST requests match the same action method, but with different parameters.
- The PUT request with the URI “api/contacts” will fail, since no correspondence is found.

3.4.3 Model

Considering the framework mentioned before, we know that it uses and adheres to some specific logic in order to provide the development of Web applications. An application that uses this kind of frameworks and makes several HTTP requests, knows that each endpoint will have to receive and handle each request in a reasonable manner. Thus, it was important in this dissertation to understand some of the underlying logic of the targeted frameworks in order to simulate an interaction between two services. It is important to recall that this work is of a basic nature and not meant to provide complete support.

We know that an HTTP-based API consists mainly of URLs, HTTP methods, and request and response formats. Based on this, it was decided that the main connection points between two partial flows of different services would be all code elements associated with the URLs and the HTTP methods.

For this particular solution, the action methods in a controller file are service entry points and the methods that execute the HTTP requests are service exit points. In both programming languages, the latter methods should preferably be related to REST, and this is the only reference to it in our solution. We do not verify that an

API satisfies any of the REST constraints, and we allow the support of methods that create HTTP requests but do not explicitly reference REST in their documentation. For example, we support methods from C# third-party libraries such as “RestSharp” or from Java frameworks like “RestTemplate”.

We know that a complete result is a flow that has to indicate a weakness in an application. This type of results have to point out attack vectors, that is, paths or means by which an attacker can cause a malicious outcome [98].

In order to produce complete flows, we have to look for unsanitized paths from source elements to sink elements. In other words, we follow basically the ideas of most queries, as highlighted in Subsection 3.2.4. Also in this model, the source elements are user inputs that enter the service layer through a controller component.

Since complete flows rely entirely on partial ones, we must first specify the latter. And so, the model considers the following three paths of a DFG as the most important partial flows:

- A) From source elements to service exit points without any type of sanitation;
- B) From a action method to a service exit point without any type of sanitation;
and
- C) From a action method to a sink element without any type of sanitation.

Note that the sanitation aspect is present in all of the three flows. Type B flows represent the transport of unsanitized user input from one service to another. Also, we seek to connect the partial flows in the following order:

$$(A \rightarrow B \rightarrow \dots \rightarrow B \rightarrow C).$$

Suppose that Figure 3.9 represents an abstract complete flow.

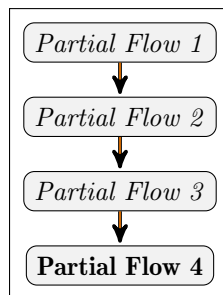


Figure 3.9: A complete flow composed of four partial flows.

Since node is synonymous with vertex in graph theory, we will simply refer to the vertices of a DFG as nodes.

Let us assume that partial flow 1 is type A, partial flows 2 and 3 are type B and partial flow 4 is type C. Although type A and type B have well established distinct classifications, in the model, these flows are represented by the same type. Currently, nothing is used to differentiate them, so it is not really known which nodes are source elements. And therefore, a source element is always considered an action method. It would be useful to know whether the data entering a controller comes from a service or not, because then we would know which action methods connect the service layer to its exterior.

Figure 3.10 shows the flow structure of type A and B. Let us suppose it represents the partial flow 2 of Figure 3.9.

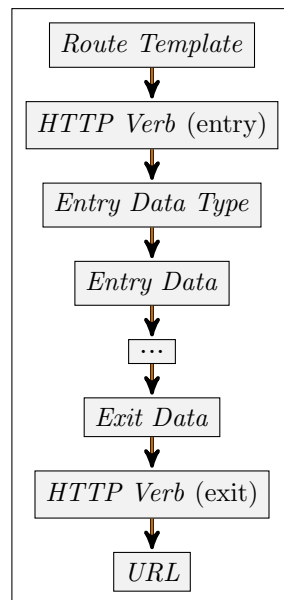


Figure 3.10: The structure of partial flow 2.

The first four nodes pertain to action methods, where:

- Node “HTTP verb (entry)” is the type of verb associated with the action method, that is, the type of request that the method handles.
- Node “entry data type” is the type of the method parameter; and
- Node “entry data” is the code element representing the method parameter.

The last three nodes pertain to HTTP requests, where:

- Node “HTTP verb (exit)” represents the type of HTTP verb;
- Node “exit data” represents the data being sent; and

- Node “URL” represents the URL to where the request is sent.

It is important to note that these figures should only be considered as illustrations, since there could be several nodes to represent a route template, a URL, etc.

Figure 3.11 depicts the flow structure of type C. Let us assume that it represents partial flow 4 of Figure 3.9.

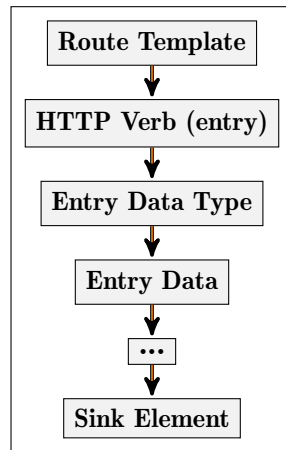


Figure 3.11: The structure of partial flow 4.

In order to produce partial flows of type A, B or C, it was necessary to identify and produce subparts of them. Thus, from now on, we will use a different notation and identify four types of partial flows: type I, type II, type III, and type IV. Additionally, we identify type V as a type of complete flows.

Flows of type I, II, and III are outputted by the new queries, and flows of type IV and V are outputted by the external tool. The relation between these five types can be summarized in the following grammar:

- $V := I^+ IV$ (type V is one or more type I followed by type IV).
- $IV := II III$ (type II followed by type III, in a particular way).

We also have that:

- Type I corresponds to a flow of type A or B; and
- The sequence II III corresponds to a flow of type C.

This grammar will serve as a reference point for the following five definitions.

Type I

The Figure 3.12 illustrates the structure behind a type I flow. As the figure indicates, these flows are equivalent to the previously mentioned type A and type B flows.

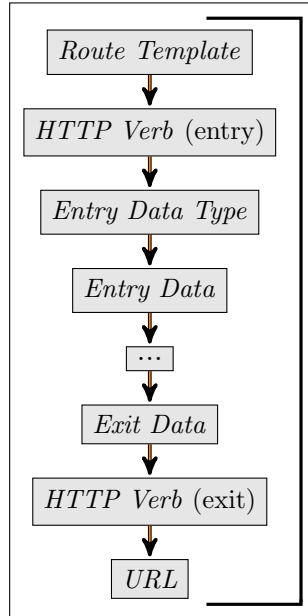


Figure 3.12: A type I flow structure.

Type IV

Figure 3.13 illustrates the structure behind a type IV flow. As the figure indicates, these flows are equivalent to the previously mentioned type C flows, and are composed of type II and type III flows.

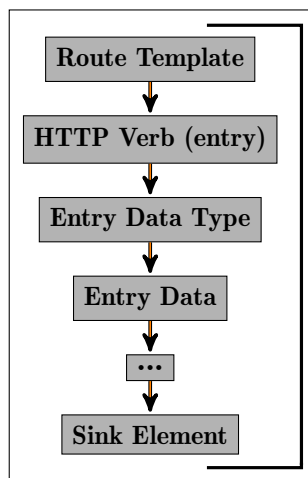


Figure 3.13: A type IV flow structure.

Type II and Type III

Figure 3.14 illustrates the structure behind type II and III flows. The shaded nodes pertain to type II, the dashed ones pertain to type III, and the “entry data” node pertains to both.

The point of connection between these two flows is the “entry data” node, and these two together form a type IV flow.

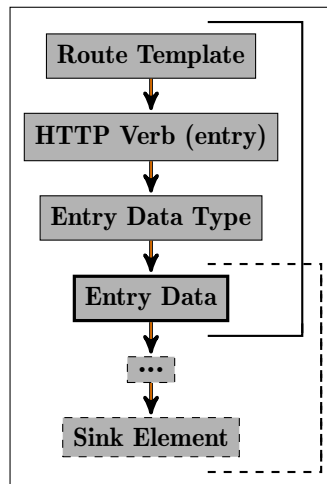


Figure 3.14: A type IV flow structure composed of a type II and type III flow.

Ideally, as we have access to type I and type IV flows, we also have access to type A, B and type C flows. Thus, we can now produce complete flows, or what we call in our solution, type V flows. These type V flows are produced by the external tool, and are therefore simply considered complete results.

Type V

Figure 3.15 illustrates the structure behind type V flows. In this figure, the shaded flows are of Type I, and the dashed flow is of type IV. This flow is considered to be a complete flow, since it is composed of type I and type IV flows from different services. It is important to note that type V flows need all of the previously mentioned types, since type IV flows are built from type II and III flows.

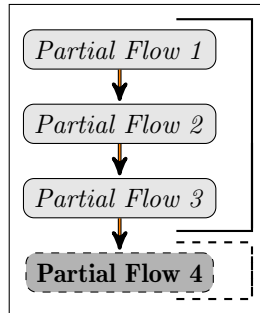


Figure 3.15: A type V flow structure composed of three type I flows and a type IV flow.

Paths Between Flows

We can use graph theory and the concept of paths in order to ease the construction of type V flows. To do this, we assume that type I and type IV flows are vertices and we try to define simple paths from type I to type IV. In this context, we refer to type IV vertices as *target vertices*, since we want to produce paths exclusively to them.

In the construction of the graph, we build edges between flows according to the connection points discussed earlier. It is important to mention that there is a graph for each vulnerability. For example, when we scan an application, we build a graph for SQL injection, another one for insecure deserialization, etc.

Using type I and type IV vertices, we want to define simple paths from type A flows to type C flows. Since type A represents connections between the service layer and its exterior, then in these graphs all type A vertices would have in-degree equal to zero. Our idea to approximate the production of paths from these flows is to produce paths from type I vertices with an in-degree equal to zero. We refer to these vertices as *entry vertices*.

We also want to define paths without providing redundant information. In this context, we intend to avoid subpaths, as we believe that they do not provide new information to what is already being shown. We also aim to disclose which services influence others, so that developers can choose where to fix the vulnerabilities or prevent their exploitation. We will give more information about this topic in Subsection 3.4.5.

Small Example

We will now present a simple example of possible interactions in an abstract microservice-based application. Let us assume that all presented interactions are related to possible SQL injection exploits.

Figure 3.16 depicts the possible interactions between nine abstract services from this application, listed alphabetically from A to I.

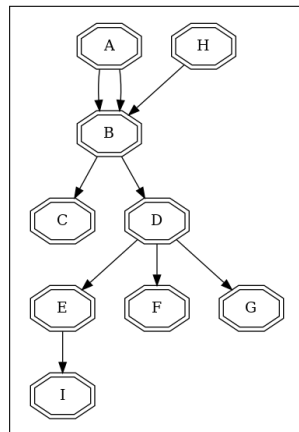


Figure 3.16: The interactions present in the small example.

Let us suppose that we applied our solution to this application and searched for SQL injection vulnerabilities. In this process, we have produced a graph, whose vertices are partial flows. In order to simplify the presentation, we have grouped each flow to the service from which it comes. For example, vertices A1 and A2 represent type I flows of service A and vertex C1(t) represents a type IV flow of service C. Figure 3.17 illustrates this graph.

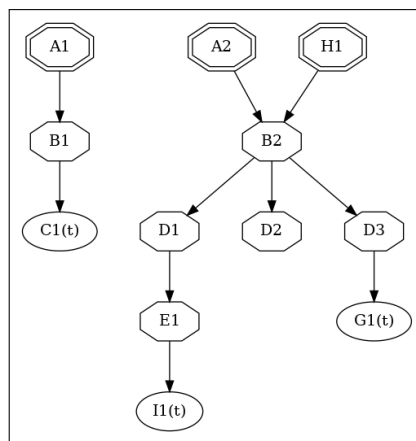


Figure 3.17: The graph of the small example.

In order to distinguish the vertices, we will only present some of their nodes. For vertices of type I, we present the route template, the HTTP verb associated with the action method, the URL and the verb of the HTTP request. For vertices of type IV, we present the route template, the HTTP verb associated with the action method, and the sink element.

In particular, vertex A1 has the following aspects:

- The route template is “URL_1A”;
- The HTTP verb that the action method handles is POST;
- The verb of the HTTP request is PUT.
- The URL of the HTTP request is “URL_1B”; and

In the case of type IV vertices, we also label them with a “(t)” at the end of their name, so that we can better distinguish them. In particular, vertex C1(t) has the following aspects:

- The route template is “URL_1C”;
- The HTTP verb that the action method handles is POST; and
- The word “End_1C” represents the SQL injection sink element.

The vertices of this example are the following:

Service A

- A1 = (URL_1A, POST, PUT, URL_1B)
- A2 = (URL_2A, PUT, PUT, URL_2B)

Service B

- B1 = (URL_1B, PUT, POST, URL_1C)
- B2 = (URL_2B, PUT, PUT, URL_1D)

Service C

- C1(t) = (URL_1C, POST, End_1C)

Service D

- D1 = (URL_1D, PUT, PUT, URL_1E)
- D2 = (URL_1D, PUT, PUT, URL_1F)

- $D3 = (\text{URL_1D}, \text{PUT}, \text{PUT}, \text{URL_1G})$

Service E

- $E1 = (\text{URL_1E}, \text{PUT}, \text{PUT}, \text{URL_1I})$

Service G

- $G1(t) = (\text{URL_1G}, \text{PUT}, \text{End_1G})$

Service H

- $H1 = (\text{URL_1H}, \text{PUT}, \text{PUT}, \text{URL_2B})$

Service I

- $I1(t) = (\text{URL_1I}, \text{PUT}, \text{End_1I})$

As mentioned before, we are looking to define paths from entry vertices to target vertices, without producing subpaths. In this particular example, the entry vertices are: A1, A2, and H1, and the target vertices are: C1(t), I1(t), and G1(t). Therefore, we want to produce the following paths:

- $\langle A1, B1, C1(t) \rangle$;
- $\langle A2, B2, D1, E1, I1(t) \rangle$;
- $\langle H1, B2, D1, E1, I1(t) \rangle$;
- $\langle A2, B2, D3, G1(t) \rangle$;
- $\langle H1, B2, D3, G1(t) \rangle$.

In this example, vertex D2 has no edges leaving it. This was done on purpose because we want to show that our solution may not recognize some interactions. In the graph, it does not exist a vertex with the route template “URL_1F” and an action method that handles HTTP PUT requests. Possible reasons for this include our solution not being able to recognize the action method, the flow being sanitized, there may have been a problem connecting type II and type III flows (if they exist), or the flow did not reach a recognizable sink element, among other things.

This example also shows that a type I flow only connects to a single action method. For example, vertex B2 is adjacent to vertices D1, D2, and D3. These three flows start from the same action method and do different things. More specifically, D1 represents an HTTP request to service E, D2 represents a request to service F, and D3 represents a request to service G.

Additional Types

In the last stages of this dissertation, two additional types of flows were created so that we could test the coverage of more cases, and in doing so, try to improve our work. We will not provide a great deal of detail about these types, as they are quite basic in comparison to others. They utilize many of the previous ideas, such as starting with route templates, and using the concept of “tainted” analysis.

With these two additional types, there exists a new way to build complete flows. They are related to HTTP GET, because this verb is not handled by the previous types.

Suppose there is an interaction between only two services where the first makes an HTTP GET request asking for some data, and the second one responds with it. In this context, we have that:

- Additional type A searches for HTTP GET requests, and binds the response data to the sink element; and
- Additional type B searches for actions methods that handles GET requests, and binds it to a return statement.

The connection points between these two flows are URLs and the HTTP verbs. In order to produce a type V flow, the external tool needs to split additional type A flow into two flows: A1 and A2. Flow A1 is from the action method to the URL of the HTTP request, and flow A2 is from the response data to the sink element. Therefore, a complete flow has the following structure: $(A1 \rightarrow B1 \rightarrow A2)$, where B1 represents an additional type B flow.

Although the connection between these two flows could, with some adaptations, be a type IV flow, we chose not to do so, in order to not influence the rest of the work. It is important to note that these additional types produce complete flows without the search for paths.

Figure 3.18 illustrates the structure behind type V flows using the additional types. The shaded nodes pertain to the additional type A flow, and the dashed ones pertain to the additional type B flow.

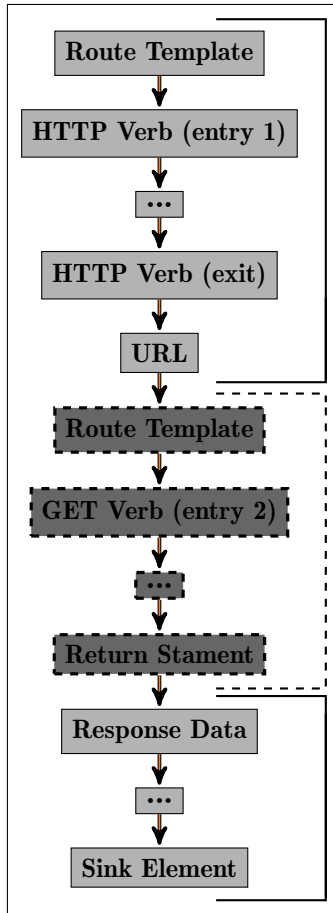


Figure 3.18: A complete flow structure composed of two additional type flows.

Having presented all types of flows, we can now describe the queries developed in this dissertation.

3.4.4 Queries

With the use of CxAudit, we developed a collection of queries specifically for microservices. These queries have strong resemblances on existing ones created by Checkmarx.

There are five types of queries: type I, type II, type III, additional type A, and additional type B, producing flows of the corresponding type.

There exists two type I queries in our solution, one for SQL injection, and another for insecure deserialization. These queries have the following structure:

1. Find service entry points;
2. Find service exit points;
3. Find places where the source code sanitizes input;

4. Define paths from service entry points to service exit points that do not pass through sanitation.

Type I queries follow a structure very similar to the ones mentioned in Subsection 3.2.4. The difference between the SQL injection query and the insecure deserialization query is mainly the sanitizing building blocks used in the path construction.

In our solution, there is only a type II query. Besides producing type II flows, it also is used to implement type I queries as a function. This is due to the evident similarities between both structures. Doing it this way avoids code repetition and also makes the query code more reader friendly.

There exists two type III queries in our solution. They are simply the standard queries created by Checkmarx and existed long before the start of this work. They are regularly updated due to the evolving nature of application security, and we use these standard queries in order to provide continuous support to this work in a simple fashion.

When given a microservice-based application, CxSAST and all these five queries are executed on it. For each flow produced we will have a scan file, which in turn is later exported to the external tool.

One aspect to note is that we will have an excess of type II and type III scan files, because we will produce flows for each action method and for each vulnerability recognized in the source code. This can be corrected through the creation of a type IV query, thereby removing the production of type II and type III flows. Unfortunately, we did not have time to fully develop or test these ideas, since these changes are not, in principle, easy to implement.

Also, we only developed queries for these two vulnerabilities mostly because of time constraints. It may take some time to create or find scenarios to test, and some vulnerabilities take into account far more details than these two. Also, we chose to implement SQL injection and insecure deserialization because they are usually mentioned as examples of OWASP Top 10 vulnerabilities found in microservices.

The work done here takes into account the support of future vulnerabilities. We looked to ease the integration of additional vulnerabilities, especially those detected by queries that follow a similar structure to the ones mentioned Subsection 3.2.4. This was apparent at a time when we only detected SQL injection vulnerabilities. In other words, with the queries we had back then, we developed the support for insecure deserialization without much effort.

The next subsection ends this section with the description of the external tool referred to as “Results Connector”.

3.4.5 Results Connector Tool

This external tool is designed to connect partial flows and produce complete flows. The results connector tool was written in C# and has the following structure:

1. Receives scan files exported from CxSAST.
2. Categorize all results in relation to flow type and vulnerability type.

For each vulnerability:

3. Connect type II and type III flows.
4. Construct a graph.
5. Define all simple paths between type I flows and type IV flows, in order to produce complete flows.
6. Produce complete flows with the additional type flows.
7. Output all the produced complete flows in the form of scan files.

More specifically:

- In the second point the tool knows, just by looking at the scan file, what is the flow type and the associated vulnerability, because this information is explicitly stated when creating these files.
- In the fourth point, we define edges between two partial flows (F_a , F_b) if they satisfy three conditions. In these three tests, we have that F_a is always of type I, and F_b is either of type I or of type IV. The three conditions are as follows:
 1. Check whether the HTTP verb of the request (from F_a) matches the HTTP verb of the action method (from F_b).
 2. Check whether the URL of the request (from F_a) matches the route template of the action method (from F_b).
 3. Check some additional details, such as issues related to class instances. We will not explain them in detail, because they are too technical and specific.
- In the fifth point, we produce paths using an adapted version of a depth-first search.

These three conditions are very important because without them we would have a large number of false positives. Suppose we have access to a partial flow that ends with a POST request, and then we connect it to another one that starts with action method that handles PUT requests. Note that in this case it would be absurd to link these two flows, and that our tool tries to prevent this connection with the enforcement of condition 1. Even so, meaningless connections like these may happen in our tool, due to the large number of scenarios that we have to support. In order to improve on this, we have made several tests on the application mentioned in Section 4.1, which we will discuss later.

From now on and until the end of this chapter, we will focus on the algorithms that are closely related to the production of paths between flows, i.e., the algorithms from the fifth point of our presented structure. We will refer to this implementation as *path production*, and we will start our description with the disclosure of seven remarks.

Remarks About Path Production

First, it is important to note that in this dissertation we have not had access to any real-world microservice-based applications. All our tests were done on GitHub samples, or on samples developed or adapted exactly for this work. Although we have a general idea on how this type of applications should be developed, we do not have any concrete data on how real-world applications are written or developed. In other words, we do not have any confirmation on the type of graphs we expect to receive. The number of vertices depends on how vulnerable the application is, because the more vulnerabilities we detect, the more partial flows we obtain, and consequently the more vertices the graphs will have. Also, the number of edges depends on how often the services interact with each other. We assume that generally we are working with sparse graphs due to the nature of loosely coupled systems. This is the main reason why we use an adjacency-list representation.

Secondly, we need to complete the definitions of entry vertex and target vertex. For the implementation, we define an *entry vertex* as a type I vertex with in-degree equal to zero and with out-degree different than zero. We also define a *target vertex* as a type IV vertex with in-degree different than zero and out-degree equal to zero. We added these restrictions because the previous definitions were not sufficient. For example, if we look for paths from a vertex with out-degree equal to zero, nothing is produced.

Thirdly, in our implementation, the vertices are numbered sequentially with integer values starting with 0. The first vertices of the sequence are the ones of type I, followed by the vertices of type IV.

Fourthly, when we refer to a set or definition of vertices/paths that satisfy an abstract condition A, we are referring to all the vertices/paths that satisfy condition A from a given abstract directed graph.

Fifthly, the pseudocode of our implementation is presented in algorithm 10. Auxiliary tasks are performed by algorithms 3-14.

Additionally, in order to clarify certain aspects of our implementation, we will present four small graphs as examples of something we want to highlight. In an attempt to make their presentation easier, we have abstracted these graphs as much as possible, and we do not present most of the information, unlike the example presented in Subsection 3.4.3.

And lastly, in the pseudocode that we will present later, we have the following details. Recalling that this external tool is written in C#, on what relates to data types:

- A vertex is of type `int`, that is, an integer type between the range of -2147483648 to 2147483647.
- A path is a list of `int`;
- The adjacency-list representation is written as an array containing lists of `int`;
- The visited and coverage attributes are written as arrays of boolean values.

On what relates to supporting functions and procedures, we have that:

- Function `length` calculates the length of a list or an array. In order to avoid ambiguity, it is important to note that in our implementation, the length of a path is not the number of edges in the path, but instead the number of vertices in the path.
- Function `Copy` creates a copy of a specified list without it pointing to the original list.
- Procedure `Push` adds a specific item to the end of a specified list.
- Procedure `Pop` removes the last item from a specified list.
- Procedure `Remove` removes a specified item from a specified list.
- Procedure `Extend` adds to the end of the first specified list all the elements of the second specified list.

Also, the notation `Initialize` means that we are adequately initializing data collections such as lists or arrays.

Having disclosed all remarks, we can now present our implementation.

Adapted Depth-First Search

The two most important algorithms in our implementation are algorithms 3 and 4, which produce all simple paths between two vertices. These algorithms are an adaptation of the basic depth-first search algorithm presented in Subsection 3.1.4.

Paul E. Black [99] states that the solution to the problem of listing all simple paths between two vertices can be achieved using a modified depth-first search. He expands on this by saying that this modified algorithm can still mark the vertices as visited in procedure DFS-VISIT, but then has to remove the mark just before returning from the recursive call. The following pseudocode is the adapted version of algorithms 1 and 2.

Algorithm 3 DFS_AllSimplePaths

```

1: function DFS_AllSimplePaths(adjLists, source, target)
2:   Initialize(path, paths)
3:   for ( $i := 0$  to Length(adjLists)) do
4:     | visited[ $i$ ] := FALSE
5:   end for
6:   DFS_AllSimplePathsAux(adjLists, source, target, visited, path, paths)
7:   return paths
8: end function

```

Algorithm 4 DFS_AllSimplePathsAux

```

1: procedure DFS_AllSimplePathsAux(adjLists, source, target, visited, path, paths)
2:   visited[source] := TRUE
3:   Push(path, source)
4:   if (source = target) then
5:     | pathCopy := Copy(path)
6:     | Push(paths, pathCopy)
7:   else
8:     for (each vert  $\in$  adjLists[source]) do
9:       | if (visited[v] = FALSE) then
10:        | DFS_AllSimplePathsAux(adjLists, vert, target, visited, path, paths)
11:       | end if
12:     end for
13:   end if
14:   Pop(path)
15:   visited[source] := FALSE
16: end procedure

```

Algorithms 3 and 4 have additional characteristics. First of all, this depth-first search does not paint vertices, but instead simply marks each vertex as visited or not visited. Secondly, it does not construct a predecessor tree, and instead produces paths while it examines all the vertices reachable from the source vertex.

The first small graph is depicted in Figure 3.19. It has seven type I vertices and one type IV vertex, this being vertex 7.

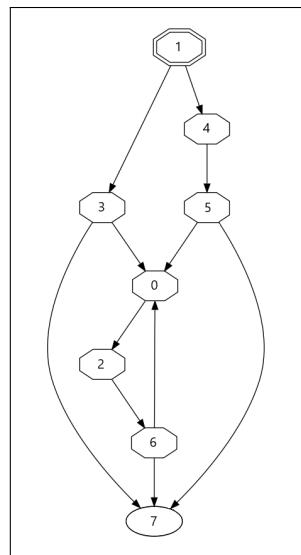


Figure 3.19: The first small graph.

Calling the **DFS_AllSimplePaths** function in order to find out all the simple paths beginning in vertex 1 and ending in vertex 7, produces the following simple paths:

- $\langle 1, 3, 0, 2, 6, 7 \rangle$;
- $\langle 1, 3, 7 \rangle$;
- $\langle 1, 4, 5, 0, 2, 6, 7 \rangle$;
- $\langle 1, 4, 5, 7 \rangle$.

From now on, in order to better present the rest of our implementation, we will divide it into three parts. The first part produces *entry paths*, the paths produced by calling the **DFS_AllSimplePaths** function for every possible entry-target pair, the second one produces *coverage paths*, and the third describes the remaining, and supportive algorithms.

Entry Paths

The first part produces a set of paths EP referred to as entry paths, which contains all simple paths from entry vertices to target vertices. Since all entry vertices have in-degree equal to zero, then EP does not contain subpaths. That is,

$$\forall p_1, p_2 \in EP . (p_1 \neq p_2) \Rightarrow (p_1 \text{ is not a subpath of } p_2 \text{ and } p_2 \text{ is not a subpath of } p_1).$$

The second small graph is depicted in Figure 3.20. It has six type I vertices and one type IV vertex, this being vertex 6.

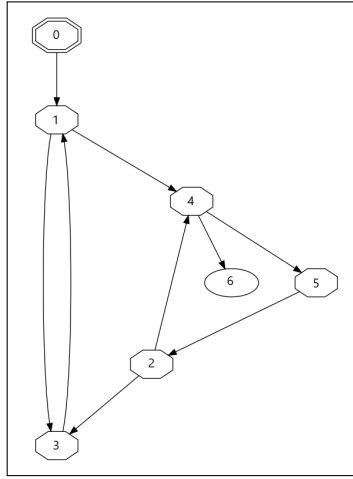


Figure 3.20: The second small graph.

For this example, we have that the **DFS_AllSimplePaths** algorithm only produces the entry path $\langle 0, 1, 4, 6 \rangle$, and that vertices 2, 3, and 5 are not present in this output. The reason why this happened was because the valid paths from 0 to 6 that pass through these vertices have loops, and thus are ignored by the **DFS_AllSimplePaths** algorithm. Since important information has been left out, there is a need to cover missing vertices and produce more paths.

From now on, we will refer to all vertices that are not contained in the entry paths as *missing vertices*. We also will refer to all paths starting from these vertices and ending in target vertices as *missing paths*. In order to determine which are the missing vertices, throughout the production of entry paths, we also update a vertex attribute called *coverage*. More specifically, for each path $p \in EP$, we see which vertices the path p contains (with the exception of entry and target vertices), and mark them as covered (as seen in lines 5-10 of algorithm 6).

Algorithms 5 and 6 produce the entry paths and update the vertex attribute coverage.

Algorithm 5 ComputeEntryPaths

```
1: function ComputeEntryPaths(adjLists, entryVerts, targetVert, coverage)
2:   Initialize(entryPaths)
3:   for each entryVert  $\in$  entryVerts do
4:     paths := DFS_AllSimplePaths(adjLists, entryVert, targetVert)
5:     Extend(entryPaths, paths)
6:     UpdateCoverage(entryVert, paths, coverage)
7:   end for
8:   return entryPaths
9: end function
```

Algorithm 6 UpdateCoverage

```
1: procedure UpdateCoverage(entryVert, paths, coverage)
2:   if (Length(path)  $\neq$  0) then
3:     coverage[entryVert] := TRUE
4:   end if
5:   for each path  $\in$  paths do
6:     for ( $i := 1$  to Length(path) - 1) do
7:       vert := path[ $i$ ]
8:       coverage[vert] := TRUE
9:     end for
10:  end for
11: end procedure
```

Coverage Paths

The second part produces a set of paths CP referred to as coverage paths, which is a subset of all missing paths. In this second part, we iterate through all target vertices, and continuously add, and remove paths from sets S_t and CP_t . The first set S_t is referred as the set of all *skip vertices related to target vertex t* , and the second set CP_t is referred as the set of all *coverage paths to target vertex t* . For the next descriptions, let T denote the set of target vertices.

There exists four important aspects to note for each abstract target vertex $t \in T$ iteration:

- We use the set M of all missing vertices.
- At the beginning of the iteration, sets S_t , and CP_t are empty.
- We add, and remove vertices from set S_t along the iteration.
- We add, and remove paths from CP_t along the iteration.

The set CP is the union of all sets $CP_t, \forall t \in T$.

Since missing vertices have in-degree different than zero, we have no guarantees that we will not output subpaths. In order to solve this problem, we have implemented an idea to reduce the number of subpaths produced. Unfortunately, we do not provide any mathematical proof that we completely avoid the production of subpaths.

In order to understand this idea, let us focus on the end of an iteration $i \in \mathbb{N}_0$ of the second part (lines 12-13 of algorithm 7). We assume that iteration i produced some missing paths to a target vertex t , and are saved in set CP_t (line 11 of algorithm 7). Let p_m be any path in CP_t . We refer to all vertices that p_m contains, except for the first and the last, as *skip vertices* related to target vertex t , and save them in set S_t (line 13 of algorithm 7).

In any iteration with target vertex $t \in T$, we refer to all vertices that are simultaneously a skip vertex and a missing vertex (vertices of set $M \cap S_t$) as *unwanted vertices*, and we refer to all paths from unwanted vertices to target vertices as *unwanted paths*.

In our idea, we assume that all unwanted paths should not be present in CP_t . Since, we do not know which are the unwanted vertices from the start of any iteration $i \in \mathbb{N}_0$, we have to continuously update CP_t (line 15 of algorithm 7), and to not produce paths from vertices of set $M \cap S_t$ (lines 7-9 in algorithm 7).

Algorithms 7, 8 and 9 implement this idea.

Algorithm 7 ComputeCoveragePaths

```

1: function ComputeCoveragePaths(adjLists, targetVert, coverage)
2:   Initialize(coveragePaths, skipVertices)
3:   for (vert := 0 to Length(coverage)) do
4:     if (coverage[vert] = TRUE) then
5:       continue;
6:     end if
7:     if (vert ∈ skipVerts) then
8:       Remove(skipVerts, vert)
9:       continue
10:    end if
11:    paths := DFS_AllSimplePaths(adjLists, vert, targetVert);
12:    Extend(coveragePaths, paths)
13:    UpdateSkipVerts(paths, skipVerts)
14:  end for
15:  coveragePaths := RemoveSubpaths(coveragePaths, skipVerts)
16:  return coveragePaths
17: end function

```

Algorithm 8 UpdateSkipVerts

```
1: procedure UpdateSkipVerts(paths, skipVerts)
2:   for each path  $\in$  paths do
3:     for ( $i := 1$  to Length(path) - 1) do
4:       vert := path[ $i$ ]
5:       if (vert  $\notin$  skipVerts) then
6:         Push(skipVerts, vert)
7:       end if
8:     end for
9:   end for
10: end procedure
```

Algorithm 9 RemoveSubpaths

```
1: function RemoveSubpaths(coveragePaths, skipVerts)
2:   for each path  $\in$  coveragePaths do
3:     if (path[0]  $\notin$  skipVerts) then
4:       Push(coveragePaths, path)
5:     end if
6:   end for
7:   return coveragePaths
8: end function
```

In order to better understand this, we will provide a simple example. Let graph $G = (V, E)$ be an abstract directed graph, with $V = \{0, 1, 2, 3, 4, 5\}$. Let us suppose that we are in the second part of our implementation, and that we are searching for paths from missing vertex 1 to target vertex 4. Let us also suppose that currently $CP_4 = \{\langle 0, 3, 4 \rangle\}$, that we just found the path $\langle 1, 0, 3, 4 \rangle$, and that we are adding it to CP_4 , producing $CP_4 = \{\langle 0, 3, 4 \rangle, \langle 1, 0, 3, 4 \rangle\}$. We know that $\langle 0, 3, 4 \rangle$ is a subpath of $\langle 1, 0, 3, 4 \rangle$, and thus that we should remove it from CP_4 .

The third small graph is depicted in Figure 3.21. It has four type I vertices and one type IV vertex, this being vertex 4.

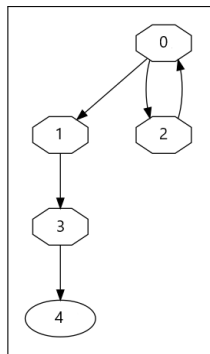


Figure 3.21: The third small graph.

The implementation of path production only produces the following simple path $\langle 2, 0, 1, 3, 4 \rangle$. This graph has no entry vertices, and because we search for coverage paths, we still have an output.

The fourth and final small graph is depicted in Figure 3.22. It has five type I vertices and two type IV vertices, these being vertices 5 and 6.

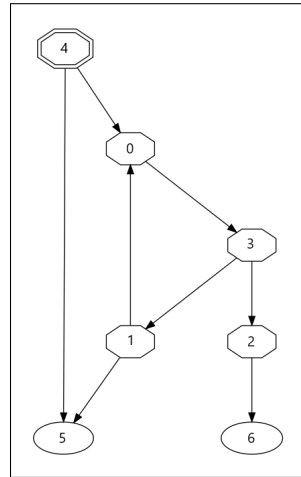


Figure 3.22: The fourth small graph.

Our implementation produces the following simple paths:

- $\langle 4, 0, 3, 1, 5 \rangle$;
- $\langle 4, 5 \rangle$;
- $\langle 4, 0, 3, 2, 6 \rangle$.

If we removed the first part of our implementation, we would have the additional simple path $\langle 1, 0, 3, 2, 6 \rangle$. In other words, if all type I vertices are marked as not covered, then the second part produces all entry paths and $\langle 1, 0, 3, 2, 6 \rangle$. The latter path states that vertex 1 influences vertex 6, which was not shown in the original three defined paths. We assume that these type of paths are residual compared to others, and that by changing our implementation to not run the first part would only increase the complexity of the algorithms without much gain.

Remaining Algorithms

Algorithms 10 and 11 connect all the other algorithms, being **ProducePaths** the main function.

Algorithm 10 ProducePaths

```
1: function ProducePaths(adjLists, numType1, numVert)
2:   Initialize(inDegree, outDegree, entryVerts, targetVerts, coverage)
3:   CalculateVertDegrees(adjLists, numType1, numVert, inDegree, outDegree)
4:   FindEntryAndTargetVerts(inDegree, outDegree, entryVerts, targetVerts)
5:   coverage := ConstructCoverage(numType1, numVert)
6:   paths := ProducePathsAux(adjLists, entryVerts, targetVerts, coverage)
7:   return paths
8: end function
```

Algorithm 11 ProducePathsAux

```
1: function ProducePathsAux(adjLists, entryVerts, targetVerts, coverage)
2:   Initialize(paths)
3:   for each targetVert  $\in$  targetVerts do
4:     targetVert := targetVerts[i]
5:     entryPaths := ComputeEntryPaths(adjLists, entryVerts, targetVert, coverage)
6:     Extend(paths, entryPaths)
7:     coveragePaths := ComputeCoveragePaths(adjLists, targetVert, coverage)
8:     Extend(paths, coveragePaths)
9:   end for
10:  return paths
11: end function
```

In the **ProducePaths** algorithm, we have that:

- The input of this algorithm is the adjacency list, the number of type I vertices, and the number of total vertices.
- Collections inDegree and outDegree are of type array of int.
- Collections entryVerts and targetVertices are of type list of int.
- Line 3 calls procedure **CalculateVertDegrees**, that calculates all vertex degrees (shown in algorithm 12).
- Line 4 calls procedure **FindEntryAndTargetVerts**, that finds which vertices are entry vertices and which vertices are target vertices (shown in algorithm 13).
- Line 5 calls function **ConstructCoverage**, that attributes all type I vertices to false and all type IV vertices to true (shown in algorithm 14).
- Line 6 calls function **ProducePathsAux**, that produces entry paths and coverage paths.
- This algorithm returns the outputted paths of line 6, so that the external tool can produce scan files.

To summarize, the implementation of path production does the following:

1. With the adjacency list, calculate the degrees of type I and type IV vertices.
2. With the degrees, calculate entry and target vertices.

For each target vertex t :

3. Produce paths from entry vertices to vertex t and store them in set EP.
4. Produce paths from missing vertices to vertex t , implementing the idea of skip vertices. The set CP_t is continuously updated with these missing paths.

After going through all target vertices:

5. Return the union of sets CP and EP.

The auxiliary algorithms 12, 13, and 14 are presented next.

Algorithm 12 CalculateVertDegrees

```

1: procedure CalculateVertDegrees(adjLists, numType1, numVert, inDegree, outDegree)
2:   for (vertT1 := 0 to numType1) do
3:     adjVerts1 := adjLists[vertT1]
4:     for ( $i := 0$  to Length(adjVerts1)) do
5:       | inDegree[adjVerts1[ $i$ ]] := (inDegree[adjVerts1[ $i$ ]] + 1);
6:     end for
7:     outDegree[vertT1] := (outDegree[vertT1] + Length(adjVerts1));
8:   end for
9:   for (vertT4 := numType1 to numVert) do
10:    adjVerts4 := adjLists[vertT4]
11:    for ( $j := 0$  to Length(adjVerts4)) do
12:      | inDegree[adjVerts4[ $j$ ]] := (inDegree[adjVerts4[ $j$ ]] + 1);
13:    end for
14:  end for
15: end procedure

```

Algorithm 13 FindEntryAndTargetVerts

```
1: procedure FindEntryAndTargetVerts(inDegree, outDegree, entryVerts, targetVerts)
2:   int numVert := Length(inDegree);
3:   int numType1 := Length(outDegree);
4:   for (i := 0 to numType1) do
5:     if (inDegree[vertT1] = 0 and outDegree[vertT1] ≠ 0) then
6:       | Push(entryVertices, vertT1);
7:     end if
8:   end for
9:   for (vertT4 := numType1 to numVert) do
10:    if (inDegree[vertT4] ≠ 0) then
11:      | Push(targetVerts, vertT4);
12:    end if
13:  end for
14: end procedure
```

Algorithm 14 ConstructCoverage

```
1: function ConstructCoverage(numType1, numVert)
2:   for (vertT1 := 0 to numType1) do
3:     | coverage[vertT1] := FALSE;
4:   end for
5:   for (vertT4 := numType1 to numVert) do
6:     | coverage[vertT4] := TRUE;
7:   end for
8:   return coverage
9: end function
```

Due to time constraints, we were unable to provide a correction or complexity analysis for the algorithms presented in this subsection. Therefore, a closer look at these two aspects can be seen as important future work.

Having described our solution, we can now present the tests we have done on it. As such, the next chapter provides a description of the case studies performed.

Chapter 4

Case Studies

This chapter presents the case studies, which have given us valuable information on what our solution is capable of doing. With this information, we have updated our solution several times and became aware of some underlying issues. The end of this chapter introduces the externalized configuration pattern, along with some ideas on how to support it. We consider worthwhile to reference this microservice pattern, as it can assume a central role in possible future work.

It is important to mention that at the time of writing, we did not have access to our solution in full (namely, to the Checkmarx tools), and thus cannot illustrate the tests made in the case studies.

4.1 Arithmetic Application

Even before we started developing the queries, we felt the need to have a collection of simple applications where we could test the various iterations of our solution. The initial search for samples of microservice-based applications was not very productive, mainly because the samples would not be easily understood by someone unfamiliar with enterprise application architecture and design. Moreover, many of the samples had the services interacting through asynchronous communication. Something not yet mentioned is that most authors of books on microservices advise that services should interact through asynchronous communication, for reasons we consider unnecessary to discuss in this dissertation.

Knowing that we wanted to start with a basic and simple support, it was decided to design an application that throughout its development would help us understand better various concepts within and outside the field of microservices. Thus, the application called Arithmetic application was devised, which would put into practice

the knowledge acquired until and after that initial phase.

This application, written in C#, simply receives input from a user and performs two arithmetic operations. This application is composed by:

- Two services;
- A simple user interface that receives user input; and
- A simple API Gateway that routes HTTP requests.

Each service contains an SQL database, and an HTTP-based API with resource representation in JSON. These APIs satisfy some REST constraints, since some of them are too demanding for a simple application. Service One calculates the addition of numbers and Service Two calculates the division and addition of numbers. More specifically, when given an list L of integers with length $n \geq 2$:

- Service One calculates its sum, that is,

$$\text{sum} = L[0] + L[1] + \dots + L[n - 1].$$

- Service Two produces list L' by sorting L by descending order, and calculates the following result:

$$\text{result} = \frac{L'[0]}{L'[1]} + \frac{L'[1]}{L'[2]} + \dots + \frac{L'[n - 2]}{L'[n - 1]}.$$

Both services can receive a list of numbers from the user interface, from the other service, or by using the contents of their respective databases. Both databases have a table composed of three columns, the first column stores numbers, the second stores strings, and the last one stores numbers.

Figure 4.1 depicts a representation of the database of service Two.

Id	ResultInFull	ResultInNumeral
1	One	1
2	Ten	10
3	One hundred	100

Figure 4.1: The database of service Two.

These applications contain several controller files distributed over the two services, and there is a support for the three HTTP verbs mentioned (GET, POST, and PUT). These services interact in several ways in order to test our solution, i.e., they give an opportunity to exploit the various vulnerabilities inserted in this application. For example, the following interaction could lead to an exploit:

1. The user interface receives data from a user.
2. The user interface sends the data through an HTTP POST request to service One.
3. Service One handles the data.
4. Service One sends the data that it received to service Two through an HTTP PUT request.
5. Service Two handles the data and updates its database.

Figure 4.2 illustrates the interaction between all of these components.

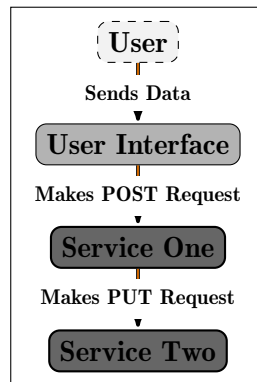


Figure 4.2: An interaction between the several components present in the Arithmetic application.

The example above can be used to test a particular exploit of SQL injection. Thinking of something similar to the example given in the Subsection 2.1.2, a malicious user can craft an SQL statement that negatively impacts the database of service Two. Figure 4.3 depicts this idea by illustrating the structure of a complete flow resulting from the exploit. In this figure, the shaded nodes are from service One, and the dashed nodes are from service Two.

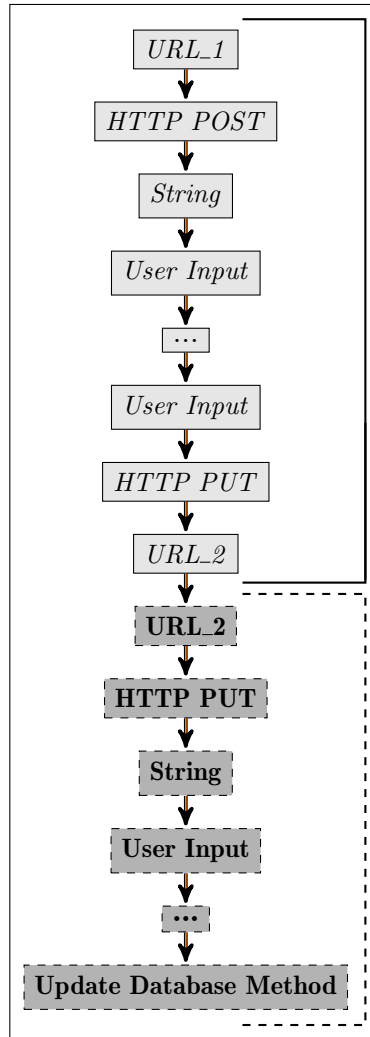


Figure 4.3: A complete flow representing an SQL injection vulnerability.

We implemented different versions of these interactions in order to test different scenarios, for instance when the application validates the user input. We used this application as a testing platform, having simulated insecure deserialization exploits such as the one presented in Subsection 2.1.2. This application was fundamental in the development of our solution, being the subject to several changes throughout the various phases of this dissertation.

4.2 Cloud Sec Application

Another application that played an important role in the development of our solution was the Cloud Sec application, developed by a colleague from Checkmarx. This application serves as the Java counterpart of the Arithmetic application, although

assuming a much smaller role. We were not informed about the purpose of the application, and we only know that it simulates an e-commerce application for some microservice project not related to ours.

Since Checkmarx requested the support of more than one programming language, we were given partial access to some source code of this application. With the help of another colleague, we modified and adapted the application to create test scenarios. In particular, we adapted a service that handles client data, and a service that handles orders from clients. The scenarios created here are similar to those of the Arithmetic application, although far less tests were performed.

4.3 Github Examples

After reaching the last stages of development in this dissertation, we directed our attention to testing our work on samples available in the GitHub platform. We searched again for representations of microservice-based applications, but this time we were better prepared for what could possibly be presented. At this stage we had a general idea on how these applications were generally designed, written, and developed.

In the end we chose several samples, such as the LAB Insurance Sales Portal project [100], the ECommerce project [101], the Manga project [102], and the spring cloud REST TCC project [103].

Due to the small amount of functionality that our solution supports, we cannot simply take a collection of GitHub samples and apply the solution to them. In order to get around this problem and test our work, we made several adaptations to the samples. The first thing we did was to find in the source code methods that create HTTP requests, and insert them into action methods. We tried to create flows that pass through controller files, while still trying to preserve some of the intended functionality of these applications. We made these changes because queries look for flows that start in action methods, and most of these applications rarely have their source code written this way.

The second change we made was to ensure that the URLs values used in the HTTP requests were hard coded. This is because we want methods that use hard coded URL values, and the addresses usually are not directly embedded in the source code of these applications.

The last change was actually done in our solution. We developed custom queries in order to support imaginary “vulnerabilities”. Since the applications are very simple

and have little functionality, we did not expect to find a reasonable number of vulnerabilities, and much less find SQL injection or insecure deserialization issues. We looked for names of methods that were influenced by other services, and we marked them as vulnerable in these new queries. We did this in order to simulate new sink elements. Naturally, we had to slightly adapt the external tool to recognize these query results.

With all these changes, we found and connected partial flows. This case study has shown that our work is much open for improvement. The following section provides an example of a potential change in our solution.

4.4 Towards Real-World Applications

This last section presents a microservice pattern that we consider important to support in future work. With the information provided here, we want to emphasize the importance of future work on top of this dissertation, and that our solution is not yet ready to be used in real-world applications.

4.4.1 Externalized Configuration Pattern

In software development, the practice of hard coding is generally considered an anti-pattern, i.e., inefficient and potentially highly counterproductive. This is because most of the time hard coded data can only be modified by editing the source code and recompiling an executable, where it might be more convenient to obtain data from external sources or generate it at runtime. Data that is hard coded usually represents immutable pieces of information [104, 105].

In a microservice-based applications, services must run in multiple environments (development, testing, quality assurance, staging, production), preferably without the modification and/or recompilation of said services. Furthermore, configuration property values depend on the environment in which a service is running in. It is disadvantageous to hard code configuration property values into a deployable service, because that would require it to be rebuilt for each environment [79, 106].

In order to solve this and other problems, applications can be developed using the externalized configuration pattern. According to this pattern, all application configuration is externalized, and configuration property values are provided to service instances at runtime. In other words, services read configuration from external sources, i.e., OS environment variables, configuration files, and so forth [79, 106].

Our solution will not produce complete flows when we receive an application that does not hard code URL values. That is, we will build graphs without edges, since we are missing the addresses of HTTP requests. In fact, partial flows instead of containing URL values, contain the names of URL variables.

Before continuing with the topic at hand, it is necessary to introduce the Docker project, which offers developers a way to implement the externalized configuration pattern.

Docker

Docker is an open-source project that allows developers to automate the deployment of their applications as portable, and self-sufficient containers [60].

An important tool to highlight from this project is the Docker Compose. This tool runs multi-container Docker applications with the help of a Compose file, which defines how the containers that make up an application are configured and is written using the Compose file format [107].

eShopOnContainers Application

We assume that a considerable part of the GitHub samples treat network locations (URL variables) as configuration properties, and apply the externalized configuration pattern. The eShopOnContainers application is an example of one of these samples, and is an open-source application developed by Microsoft. Figure 4.4 illustrates its various components.

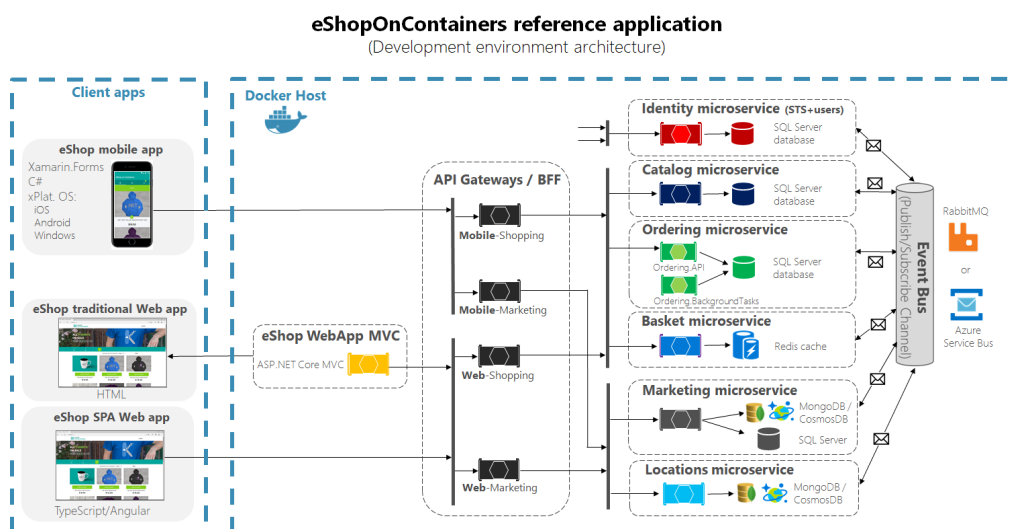
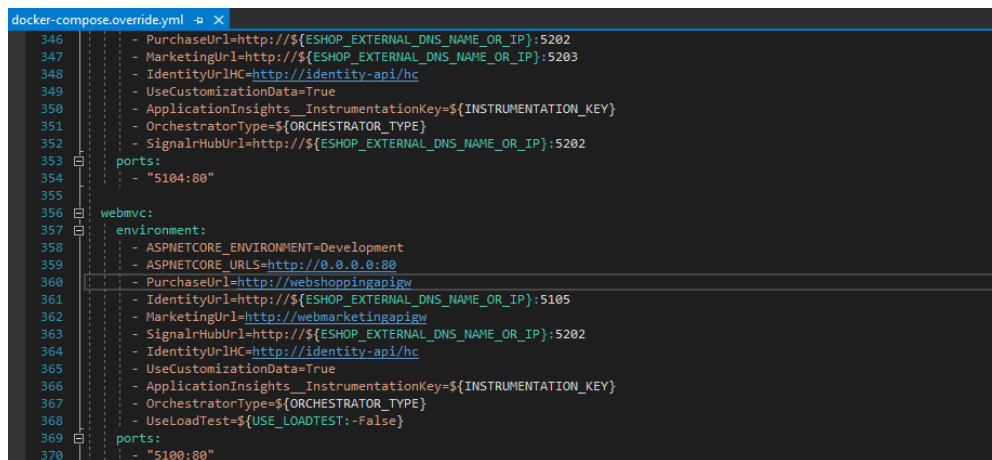


Figure 4.4: eShopOnContainers architecture overview [108].

The application is mainly written in C# and consists of several subsystems. It includes a variety of user interfaces, API Gateways, and services. The client applications interact with the services through HTTP, and the services interact with each other through asynchronous communication [108].

We mention this application because it is a good reference point for microservice architectural patterns. More importantly, most of the application is designed using Docker containers, and the application is run using Docker Compose.

Figure 4.5 shows part of the source code of a configuration file called “docker-compose.override.yml”. From what we have seen, at runtime, this application is provided with OS environment variable values. The “PurchaseUrl” variable highlighted in this figure is one of these variables and is configured with the help of Docker Compose. Although this variable is never used by a service, it is used by a Docker container, which gives us an idea of how this application applies the externalized configuration pattern. In particular, the value of the “PurchaseUrl” environment variable is part of an HTTP request address.



```
docker-compose.override.yml
346 - PurchaseUrl=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5202
347 - MarketingUrl=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5203
348 - IdentityUrlIHC=http://identity-api/hc
349 - UseCustomizationData=True
350 - ApplicationInsights_InstrumentationKey=${INSTRUMENTATION_KEY}
351 - OrchestratorType=${ORCHESTRATOR_TYPE}
352 - SignalrHubUrl=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5202
353 ports:
354 - "5104:80"
355
356 webmvc:
357 environment:
358 - ASPNETCORE_ENVIRONMENT=Development
359 - ASPNETCORE_URLS=http://0.0.0.0:80
360 - PurchaseUrl=http://webshoppingapigw
361 - IdentityUrl=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
362 - MarketingUrl=http://webmarketingapigw
363 - SignalrHubUrl=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5202
364 - IdentityUrlIHC=http://identity-api/hc
365 - UseCustomizationData=True
366 - ApplicationInsights_InstrumentationKey=${INSTRUMENTATION_KEY}
367 - OrchestratorType=${ORCHESTRATOR_TYPE}
368 - UseLoadTest=${USE_LOADTEST:-False}
369 ports:
370 - "5100:80"
```

Figure 4.5: A compose file from the eShopOnContainers application [109].

A More Complete Solution

Since our solution only extracts information from the source code of services, there is a need to adapt our work for applications that use Docker Compose. This section has shown us that information present in configuration files, such as Compose files, is very important to produce complete flows, and is missing. Although these files are not taken into account, they are actually shipped with the application.

The first idea we put forward in order to support these cases is to scan the source code of the configuration files with the use of an additional external tool. This tool may or may not be related to CxSAST.

The second idea is that the additional external tool can produce a list of key:value pairs. The keys are the variables names present in the services, and the values are the configuration property values present in Compose files. Since the variables of services are assigned with configuration values, we hope that something similar can be done by looking at the source code of configuration files. We believe that these files have both the values and the variable names, and with this we hope to produce a list of key:value pairs that gives us an idea on what the application is provided at runtime. With the list, we should be able to update the partial flows, replacing the variable names with their corresponding values, just before we create the graphs.

Figure 4.6 provides an illustration of how these ideas could be integrated with our current solution, where the left side represents our current solution, and the right side is related to the externalized configuration pattern just mentioned.

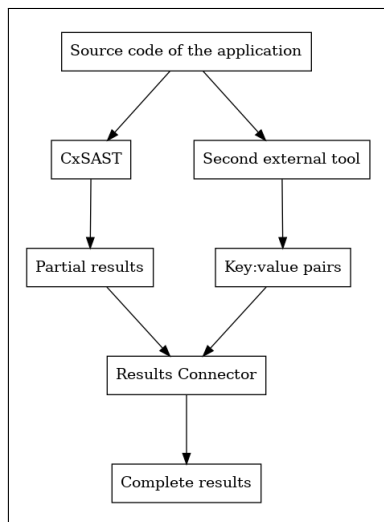


Figure 4.6: A draft of a more complete solution.

It is important to note that the second idea is not simple to implement, because we may encounter several obstacles, such as having to check several files in order to produce a single key:value pair. It is also important to note that there are several ways to configure an application in addition to the one mentioned here. Since we have not had time to investigate or test any of these ideas, a solution for the externalized configuration pattern can be seen as important future work.

Chapter 5

Conclusion

At the time of writing, microservices were becoming an important trend in software development. It is expected that more and more organizations will adhere to this architectural style, as companies like Amazon and Netflix have successfully done before. Time will tell if microservices are in fact the future direction for software architectures, but what we know now is that there exists an increasing demand for security solutions that identify vulnerabilities in microservice-based applications.

This dissertation seeks to provide a basis for one of these solutions. A solution focused on service interactions, and intended for applications that make use of multiple HTTP-based RESTful APIs. The vulnerabilities that arise from these service interactions need to be detected because, like most vulnerabilities, they may cause serious problems.

Our solution was achieved through the development of a new group of CxSAST queries and a new external tool, called Results Connector. Queries search for partial flows, and the external tool connects them together in order to produce complete flows. These two components of our solution do not support all types of services that communicate via the HTTP protocol, as this would be very difficult to do with the time we had available. Our solution is concerned with a particular type of services, such as those that make use of controller files as entry points. We also only support two programming languages (C# and Java) and two vulnerabilities (SQL injection and insecure deserialization).

We designed three types of queries: type I, II, and III, in order to separate concerns. Each has a particular role, and helps us to achieve the two desired partial flows (type I and IV). There are also additional type A and B queries, but these are concerned with requests made with HTTP GET. The latter are very basic, have strong similarities with the other ones, and were developed primarily to help future

work.

As mentioned before, the Results Connector tool has the sole purpose of connecting flows. Besides the possibility of queries producing false positives, this external tool can connect flows that should never be connected, thus producing additional false positives. In order to mitigate this, there exists three conditions in charge of verifying whether two partial flows represent a possible interaction between two services. The first condition is related to the type of HTTP verb, the second is concerned with the service addresses, and the third is relative to additional issues, such as programming class related matters. Once these conditions are checked, an adapted version of a depth-first search is used to produce simple paths between flows. Although this version searches for all simple paths between two vertices, we have implemented several techniques that reduce the number of searches performed. One of them was a small idea for reducing the number of subpaths produced. There is probably a more elegant way to solve this problem in its entirety, but our constrained search effort in the literature did not lead us to better a technique.

We consider our solution to be basic and entry-level. After seeing examples from the GitHub platform, we can say that there is the possibility for a lot of future work, as shown in Section 4.4. In the queries, there is room to primarily optimize the code, develop coverage of more cases, and support more vulnerabilities and programming languages. In the external tool, there is the possibility to optimize the algorithms used (mainly in terms of time and space), create more conditions, and revise the latter. An additional aspect will be to improve the way results are presented to customers. Another line of work will be to investigate the correctness and complexity of the algorithms used in the production of simple paths, as previously mentioned in Subsection 3.4.5.

In conclusion, Checkmarx can benefit from our work in one of two ways. They can continue forward with our solution, adapting it but maintaining some of its fundamentals (such as the flow/query model), or draw on some of our ideas, tests, and knowledge in order to design a new solution far more complete and versatile. Hence, we hope that software development experts can, after evaluating our work, start at a higher level than when we started this dissertation. There is also the matter of asynchronous communication, which is generally suggested as the ideal way for services to interact. We have explored very little on this topic, but perhaps something similar to what is shown here can also be achieved.

Bibliography

- [1] Hannousse, A. and Yahiouche, S. (2020). Securing Microservices and Microservice Architectures: A Systematic Mapping Study. arXiv:2003.07262v2 [cs.CR].
- [2] Lewis, J. and Fowler, M. (2014). Microservices - a definition of this new architectural term. <https://martinfowler.com/articles/microservices.html>. Accessed 24 January 2021.
- [3] Newman, S. (2015). *Building microservices - designing fine-grained systems, 1st Edition*. O'Reilly.
- [4] OWASP Top Ten 2017 - Foreword. <https://owasp.org/www-project-top-ten/2017/Foreword>. Accessed 24 January 2021.
- [5] Nuseibeh, R. (2012). Application Security. <https://www.checkmarx.com/glossary/application-security>. Accessed 24 January 2021.
- [6] Stark, J. CxSAST Overview. (2020). <https://checkmarx.atlassian.net/wiki/spaces/KC/pages/59211846/CxSAST+Overview>. Accessed 24 January 2021.
- [7] Application security. (2020). https://en.wikipedia.org/wiki/Application_security. Accessed 24 January 2021.
- [8] Database. (2021). <https://en.wikipedia.org/wiki/Database>. Accessed 24 January 2021.
- [9] Liu, L. and Özsu, M. T. (2018). *Encyclopedia of Database Systems, 2nd Edition*. Springer.
- [10] Serialization. (2021). <https://en.wikipedia.org/wiki/Serialization>. Accessed 24 January 2021.
- [11] Nuseibeh, R. Security Vulnerability. (2012). <https://www.checkmarx.com/glossary/security-vulnerability>. Accessed 24 January 2021.

- [12] OWASP Top Ten. <https://owasp.org/www-project-top-ten>. Accessed 24 January 2021.
- [13] OWASP Top Ten 2017 - 2017 Top 10. https://owasp.org/www-project-top-ten/2017/Top_10. Accessed 24 January 2021.
- [14] Injection Flaws. https://owasp.org/www-community/Injection_Flaws. Accessed 23 January 2021.
- [15] SQL injection. <https://portswigger.net/web-security/sql-injection>. Accessed 23 January 2021.
- [16] SQL Injection. https://owasp.org/www-community/attacks/SQL_Injection. Accessed 23 January 2021.
- [17] SQL injection. (2021). https://en.wikipedia.org/wiki/SQL_injection. Accessed 23 January 2021.
- [18] SQL Injection Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html. Accessed 23 January 2021.
- [19] CWE-502: Deserialization of Untrusted Data. <https://cwe.mitre.org/data/definitions/502.html>. Accessed 23 January 2021.
- [20] Deserialization of untrusted data. https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data. Accessed 23 January 2021.
- [21] Stöckli, P. (2017). How to configure Json.NET to create a vulnerable web API. <https://www.alphabot.com/security/blog/2017/net/How-to-configure-Json.NET-to-create-a-vulnerable-web-API.html>. Accessed 23 January 2021.
- [22] OWASP Top Ten 2017 - A8:2017-Insecure Deserialization. https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization. Accessed 23 January 2021.
- [23] Gartner Magic Quadrant for Application Security Testing. (2020). <https://www.gartner.com/en/documents/3984345>. Accessed 23 January 2021.

- [24] Brucker, A. and Sodan, U. (2014). Deploying static application security testing on a large scale. In: Katzenbeisser, S., Lotz, V. and Weippl, E. (Hrsg.), *Sicherheit 2014 – Sicherheit, Schutz und Zuverlässigkeit*. Bonn: Gesellschaft für Informatik e.V.. (S. 91-92).
- [25] Test Statistics. https://groups.bme.gatech.edu/groups/biml/resources/useful_documents/Test_Statistics.pdf. Accessed 23 January 2021.
- [26] Network. <https://dictionary.cambridge.org/pt/dicionario/ingles/network>. Accessed 23 January 2021.
- [27] Distributed computing. (2020). https://en.wikipedia.org/wiki/Distributed_computing. Accessed 15 January 2021.
- [28] Berners-Lee, T., Fielding, R. T. and Masinter, L. (2005). Uniform Resource Identifier (URI): Generic Syntax. *RFC*, 3986: 1-3.
- [29] URL. (2021). <https://en.wikipedia.org/wiki/URL>. Accessed 23 January 2021.
- [30] Interface. <https://www.merriam-webster.com/dictionary/interface>. Accessed 23 January 2021.
- [31] Interface (computing). (2020). [https://en.wikipedia.org/wiki/Interface_\(computing\)](https://en.wikipedia.org/wiki/Interface_(computing)). Accessed 23 January 2021.
- [32] API. (2021). <https://en.wikipedia.org/wiki/API>. Accessed 23 January 2021.
- [33] Client-server model. (2021). https://en.wikipedia.org/wiki/Client-server_model. Accessed 23 January 2021.
- [34] Client-side. (2020). <https://en.wikipedia.org/wiki/Client-side>. Accessed 23 January 2021.
- [35] Server-side. (2020). <https://en.wikipedia.org/wiki/Server-side>. Accessed 23 January 2021.
- [36] User Interface. https://techterms.com/definition/user_interface. Accessed 23 January 2021.
- [37] Communication protocol. (2021). https://en.wikipedia.org/wiki/Communication_protocol. Accessed 23 January 2021.

- [38] Request–response. (2020). <https://en.wikipedia.org/wiki/Request-response>. Accessed 23 January 2021.
- [39] Pattern: Messaging. <https://microservices.io/patterns/communication-style/messaging.html>. Accessed 23 January 2021.
- [40] Representational state transfer. (2020). https://en.wikipedia.org/wiki/Representational_state_transfer. Accessed 16 December 2020.
- [41] What is REST. <https://restfulapi.net>. Accessed 23 January 2021.
- [42] Fielding, R. T. and Reschke, J. F. (2014). Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. *RFC*, 7230: 1–12.
- [43] Fielding, R. T., Gettys, J., Mogul, J. C., Nielsen, H.F., Masinter, L., Leach, P. L. and Berners-Lee, T. (1999). Hypertext Transfer Protocol - HTTP/1.1. *RFC*, 2616: 1-10.
- [44] HTTP Messages. (2020). <https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages>. Accessed 23 January 2021.
- [45] Fielding, R. T., and Reschke, J. F. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. *RFC*, 7231: 21–22.
- [46] HTTP request methods. (2020). <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>. Accessed 23 January 2021.
- [47] REST - PUT vs POST. <https://restfulapi.net/rest-put-vs-post>. Accessed 23 January 2021.
- [48] An overview of HTTP. (2020). <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>. Accessed 23 January 2021.
- [49] Communication in a microservice architecture. (2020). <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture>. Accessed 23 January 2021.
- [50] Web API. (2021). https://en.wikipedia.org/wiki/Web_API. Accessed 23 January 2021.

- [51] Application software. (2021). https://en.wikipedia.org/wiki/Application_software. Accessed 23 January 2021.
- [52] What is virtualization. <https://opensource.com/resources/virtualization>. Accessed 23 January 2021.
- [53] Operating system. (2020). https://en.wikipedia.org/wiki/Operating_System. Accessed 23 January 2021.
- [54] Kernel (operating system). (2021). [https://en.wikipedia.org/wiki/Kernel_\(operating_system\)](https://en.wikipedia.org/wiki/Kernel_(operating_system)). Accessed 23 January 2021.
- [55] Firesmith, D. (2017). Virtualization via Virtual Machines. https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-virtual-machines.html. Accessed 23 January 2021.
- [56] Containers vs. VMs: What's the Difference. (2020). <https://www.ibm.com/cloud/blog/containers-vs-vms>. Accessed 23 January 2021.
- [57] Hypervisor. (2021). <https://en.wikipedia.org/wiki/Hypervisor>. Accessed 23 January 2021.
- [58] Kubernetes vs. Docker: Why Not Both. (2020). <https://www.ibm.com/cloud/blog/kubernetes-vs-docker>. Accessed 23 January 2021.
- [59] What is a Container. <https://www.docker.com/resources/what-container>. Accessed 23 January 2021.
- [60] What is Docker. (2018). <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined>. Accessed 23 January 2021.
- [61] What is container orchestration. <https://www.redhat.com/en/topics/containers/what-is-container-orchestration>. Accessed 23 January 2021.
- [62] What is cloud computing. <https://azure.microsoft.com/en-us/overview/what-is-cloud-computing>. Accessed 23 January 2021.
- [63] Loose coupling. (2021). https://en.wikipedia.org/wiki/Loose_coupling. Accessed 23 January 2021.
- [64] What is DevOps. https://aws.amazon.com/devops/what-is-devops/?nc1=h_ls. Accessed 23 January 2021.

- [65] Fowler, M. (2013). ContinuousDelivery. <https://martinfowler.com/bliki/ContinuousDelivery.html>. Accessed 23 January 2021.
- [66] Fowler, M. (2020). DomainDrivenDesign. <https://martinfowler.com/bliki/DomainDrivenDesign.html>. Accessed 23 January 2021.
- [67] What is Continuous Delivery. https://aws.amazon.com/devops/continuous-delivery/?nc1=h_ls. Accessed 23 January 2021.
- [68] DevOps. (2021). <https://en.wikipedia.org/wiki/DevOps>. Accessed 23 January 2021.
- [69] CI/CD. (2021). <https://en.wikipedia.org/wiki/CI/CD>. Accessed 23 January 2021.
- [70] What is DevOps. <https://azure.microsoft.com/en-us/overview/what-is-devops>. Accessed 23 January 2021.
- [71] Domain-Driven Design: What is it and how do you use it. (2017). <https://airbrake.io/blog/software-design/domain-driven-design>. Accessed 23 January 2021.
- [72] Fowler, M. (2014). BoundedContext. <https://martinfowler.com/bliki/BoundedContext.html>. Accessed 23 January 2021.
- [73] Domain-driven design. (2020). https://en.wikipedia.org/wiki/Domain-driven_design. Accessed 23 January 2021.
- [74] Granularity. (2020). <https://en.wikipedia.org/wiki/Granularity>. Accessed 23 January 2021.
- [75] The API gateway pattern versus the Direct client-to-microservice communication. (2021). <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern>. Accessed 23 January 2021.
- [76] Using API gateways in microservices. (2018). <https://docs.microsoft.com/en-us/azure/architecture/microservices/design/gateway>. Accessed 23 January 2021.

- [77] Yarygina, T. and Bagge, A. H. (2018). Overcoming security challenges in microservice architectures. In: *IEEE Symposium on Service-Oriented System Engineering, SOSE 2018, Bamberg, Germany, March 26-29, 2018*: 11-14.
- [78] Zimmermann, O. (2017). Microservices tenets. *Comput. Sci. Res. Dev.*, 32(3-4): 301–304.
- [79] Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning Publications.
- [80] Message Brokers. (2020). <https://www.ibm.com/cloud/learn/message-brokers>. Accessed 23 January 2021..
- [81] Yu, D., Jin, Y., Zhang, Y. and Zheng, X. (2019). A survey on security issues in services communication of microservices-enabled fog applications. *Concurr. Comput. Pract. Exp.*, 31(22): 1-12.
- [82] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press.
- [83] Kruse, R.L. and Ryba, A.L. (1998). *Data structures and program design in C++*. Prentice Hall.
- [84] Big O notation. (2021). https://en.wikipedia.org/wiki/Big_O_notation. Accessed 15 January 2021.
- [85] Amortized analysis. (2020). https://en.wikipedia.org/wiki/Amortized_analysis. Accessed 23 January 2021.
- [86] Control-flow graph. (2020). https://en.wikipedia.org/wiki/Control-flow_graph. Accessed 23 January 2021.
- [87] Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5(7): 1–2.
- [88] Data-flow analysis. (2020). https://en.wikipedia.org/wiki/Data-flow_analysis. Accessed 23 January 2021.
- [89] Dewhurst, R. Static Code Analysis. https://owasp.org/www-community/controls/Static_Code_Analysis#Taint_Analysis. Accessed 23 January 2021.
- [90] Data-Flow Graphs. <http://bears.ece.ucsb.edu/research-info/DP/dfg.html>. Accessed 15 January 2021.

- [91] CxAudit Overview. (2020). <https://checkmarx.atlassian.net/wiki/spaces/KC/pages/5406733>. Accessed 23 January 2021.
- [92] Query Structure. (2020). <https://checkmarx.atlassian.net/wiki/spaces/KC/pages/5406747/Query+Structure>. Accessed 23 January 2021.
- [93] Scan Results Example (v8.9.0 and up). (2020). <https://checkmarx.atlassian.net/wiki/spaces/KC/pages/1170441768/Scan+Results+Example+v8.9.0+and+up>. Accessed 23 January 2021.
- [94] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. In: Mazzara, M. and Meyer, B. (eds) *Present and Ulterior Software Engineering*. Springer, Cham. https://doi.org/10.1007/978-3-319-67425-4_12.
- [95] Model-view-controller. (2021). <https://en.wikipedia.org/wiki/Model-view-controller>. Accessed 23 January 2021.
- [96] Wasson, M. (2018). Routing in ASP.NET Web API. <https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api>. Accessed 23 January 2021.
- [97] HTTP routing. <https://www.playframework.com/documentation/2.8.x/JavaRouting>. Accessed 23 January 2021.
- [98] Attack vector. (2012). <https://searchsecurity.techtarget.com/definition/attack-vector>. Accessed 15 January 2021.
- [99] Black, E. P. (2008). All simple paths. <https://xlinux.nist.gov/dads/HTML/allSimplePaths.html>. Accessed 23 January 2021.
- [100] ASCLAB, LAB Insurance Sales Portal, (2020), GitHub repository, <https://github.com/asc-lab/dotnetcore-microservices-poc>. Accessed 14 December 2020.
- [101] Włodek P., ECommerce Application, (2019), GitHub repository, <https://github.com/pwlodek/ECommerce.Microservices>. Accessed 14 December 2020.
- [102] Paulovich I., Manga, (2020), GitHub repository, <https://github.com/ivanpaulovich/clean-architecture-manga>. Accessed 14 December 2020.

- [103] Chris, REST TCC, (2020), GitHub repository, <https://github.com/prontera/spring-cloud-rest-tcc>. Accessed 14 December 2020.
- [104] Hard coding. (2020). https://en.wikipedia.org/wiki/Hard_coding. Accessed 03 January 2021.
- [105] Anti-pattern. (2021). <https://en.wikipedia.org/wiki/Anti-pattern>. Accessed 03 January 2021.
- [106] Pattern: Externalized configuration. <https://microservices.io/patterns/externalized-configuration.html>. Accessed 03 January 2021.
- [107] Docker, compose, (2021), GitHub repository, <https://github.com/docker/compose>. Accessed 03 January 2021.
- [108] Dotnet-architecture, eShopOnContainers, (2021), GitHub repository, <https://github.com/dotnet-architecture/eShopOnContainers>. Accessed 03 January 2021.
- [109] Dotnet-architecture, eShopOnContainers, (2021), GitHub repository, <https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/docker-compose.override.yml>. Accessed 03 January 2021.