

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Attack Framework for Kafka-based Event-driven Architectures

Nuno Pereira



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. André Restivo

Co-Supervisor: João Dias

July 27, 2025

Attack Framework for Kafka-based Event-driven Architectures

Nuno Pereira

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Ademar Aguiar

Supervisor: Prof. André Restivo

Referee: Prof. Ana Regina Sousa

July 27, 2025

Abstract

Nowadays the entire world is connected within itself. Not only can we sense a growth in the use of the internet as a whole, but also of cyber-threats, threatening to disrupt normal day-to-day use of software systems. Because of this, knowing what kinds of systems we use and what vulnerabilities they might have is crucial. The most used type of system is synchronous in nature, where participants have to wait for one another to complete information exchanges. These kinds of systems, being simpler, present a much simpler and because of that a much more studied vulnerability landscape than their counterparts, which are asynchronous in the sense that the involved parties do not need to wait for one another to communicate. One such example of the latter type of system are *Event-Driven Systems* which, as mentioned, lack security and vulnerability research.

Reviewing what research has been done by academia revealed a major research gap in the area of vulnerabilities in Event-Driven Systems. Furthermore, historical analysis indicated that there was no trend in growing or lack of interest in researching this topic. Several efforts exist to document and research adjacent topics but none of those focus solely on Event-Driven Systems.

To expand research into this topic, in this dissertation we propose a tripartite solution, consisting of a: 1) taxonomy on vulnerabilities in Event-Driven Systems; 2) a Threat Model for Event-Driven Systems, specially those based on Apache Kafka; 3) *Opfuzz*, a tool capable of fuzzing policies developed for the Open Policy Agent (OPA) platform and possibly uncover some flaw with the policy's implementation. The taxonomy was built from a corpus of 17 articles, covering topics ranging from unauthorized system access to misconfiguration of security policies. 4 main categories of vulnerabilities were discovered: *Broken Access*, *Deserialization Issues*, *Policy Misconfiguration* and *Security Misconfiguration*. The Threat Model, developed based on the previously defined vulnerability categories, involved attempting actual exploits against vulnerable (albeit controlled) targets in attempts to validate the categories defined in the taxonomy. Lastly, *Opfuzz*, for fuzzing authorization policies, is capable of reading as input a Rego file, figuring out the shape of its input and generating values according to that shape. A report is generated which was then sent to a domain expert for manual analysis.

The results of the Threat Model show that the categorization efforts resulted in correct and valid categories, representing common vulnerabilities in the Event-Driven vulnerability landscape. Results of the validation report generated by *Opfuzz* show that it helped uncover previously unknown errors in an example Rego policy, attesting to the tool's capability of helping uncover more kinds of vulnerabilities.

Keywords: Vulnerability Taxonomy. Threat Model. Apache Kafka. Vulnerability Detection. Event-Driven Systems.

Resumo

Hoje em dia, o mundo inteiro está interligado pela Internet. Consegue-se sentir um crescimento no uso da internet como um todo, mas também um aumento nas ameaças cibernéticas vividas diariamente, que ameaçam interromper o uso normal e cotidiano de sistemas de software aos quais nos habituamos. Por conta disso, conhecer os tipos de sistemas que utilizamos e quais vulnerabilidades eles podem ter é fundamental. O tipo de sistema mais utilizado é de natureza síncrona, onde os participantes esperam por uma resposta de outro participante para completar trocas de informação. Esses tipos de sistemas, por serem mais simples, apresentam um leque de vulnerabilidades também mais simples e, por isso, muito mais estudado do que seus equivalentes assíncronos, nos quais as partes envolvidas não precisam de esperar umas pelas outras para comunicarem entre si. Um exemplo deste último tipo de sistema são os Sistemas Baseados em Eventos (*Event-Driven Systems*), os quais, como já mencionado, carecem de pesquisas aprofundadas no que toca a segurança e vulnerabilidades.

Ao rever o que foi investigado pela academia, foi revelada uma grande lacuna de pesquisa na área de vulnerabilidades em Sistemas Baseados em Eventos. Além disso, uma análise histórica indicou que não há uma tendência de crescimento nem de queda no interesse por este tema. Existem vários esforços para documentar e pesquisar tópicos adjacentes, mas nenhum deles foca exclusivamente nos Sistemas Baseados em Eventos.

Para expandir a pesquisa neste tema, nesta dissertação propomos uma solução tripartida, composta por: 1) uma taxonomia sobre vulnerabilidades em Sistemas Baseados em Eventos; 2) um Modelo de Ameaças para Sistemas Baseados em Eventos, especialmente aqueles baseados em Apache Kafka; 3) Opfuzz, uma ferramenta capaz de aplicar *fuzzing* em políticas desenvolvidas para a plataforma *Open Policy Agent* (OPA) e, possivelmente, revelar falhas na implementação das mesmas. A taxonomia foi construída a partir de um corpo de 17 artigos, abordando temas que vão desde acessos não autorizado a sistemas até erros de configuração de políticas de segurança. Foram descobertas quatro categorias principais de vulnerabilidades: Acesso Não Autorizado (*Broken Access*), Problemas de Desserialização (*Deserialization Issues*), Configuração Incorreta de Políticas (*Policy Misconfiguration*) e Configuração Incorreta de Segurança (*Security Misconfiguration*). O Modelo de Ameaças, desenvolvido com base nas categorias de vulnerabilidades previamente definidas, envolveu tentativas reais de exploração contra alvos vulneráveis (ainda que controlados), com o objetivo de validar as categorias definidas na taxonomia. Por fim, o Opfuzz, para efetuar *fuzzing* de políticas de autorização, lê como entrada um arquivo Rego, identifica o formato da *input* esperado e gera valores de acordo com esse formato. Um relatório é gerado e, posteriormente, enviado a um especialista da área para análise manual.

Os resultados do Modelo de Ameaças mostram que os esforços de categorização resultaram em categorias corretas e válidas, representando vulnerabilidades comuns no panorama de Sistemas Baseados em Eventos. Os resultados do relatório de validação gerado pelo Opfuzz mostram que ele ajudou a revelar erros antes desconhecidos em uma política Rego de exemplo, atestando a capacidade da ferramenta de ajudar a descobrir outros tipos de vulnerabilidades.

Palavras-chave: Taxonomia de Vulnerabilidades. Modelo de Ameaças. Apache Kafka. Detecção de Vulnerabilidades. Sistemas Orientados a Eventos.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

In the context of this dissertation, only one Sustainable Development Goals was addressed:

SDG 9 Industry, Innovation and Infrastructure by striving to build better and more secure systems which can more confidently move forward our technological world.

SGD	Target	Contribution	Performance Indicators and Metrics
9	9.1	Guaranteeing that Event-Driven architectures and corresponding infrastructure are secure against current and future threats to digital innovation	Increase in confidence of an digital infrastructure's security posture

“... you are only quite a little fellow in a wide world after all!”

J. R. R. Tolkien, *The Hobbit*

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation and objectives	2
1.3	Document Structure	3
2	Background	5
2.1	Request-Response Communication Model	5
2.2	Event-Driven Communication Model	6
2.3	Apache Kafka	9
3	State of the Art	11
3.1	Methodology	11
3.1.1	Databases	12
3.1.2	Publication search	12
3.1.3	Filtering	12
3.2	Results	13
3.3	Discussion	18
3.4	Evolutionary Analysis	20
3.5	Summary	20
4	Problem Statement	21
4.1	Problem	21
4.2	Hypothesis	22
4.3	Research Questions	23
4.4	Summary	23
5	Proposed Solution	24
5.1	EDA Vulnerability Taxonomy	25
5.1.1	Context	25
5.1.2	Methodology	26
5.1.3	Results	27
5.1.4	Categorization	34
5.1.5	Threats to Validity	38
5.1.6	Summary	38
5.2	EDA Vulnerability Threat Model	39
5.2.1	Context	39
5.2.2	Test Environment	40
5.2.3	Summary	45

5.3	Policy Vulnerability Detection Tooling	46
5.3.1	Rationale	46
5.3.2	System Components	48
5.3.3	Summary	49
5.4	Summary	51
6	Validation	52
6.1	Context	52
6.2	Validation Goals	53
6.3	Results	54
6.3.1	Attack Experiments	54
6.3.2	Validation Report Analysis	55
6.4	Discussion	58
6.4.1	Threats to validity	59
6.5	Summary	60
7	Conclusion	61
7.1	Discussion	61
7.2	Contributions	63
7.3	Limitations	63
7.4	Future Work	64
	References	66
A	Event-Driven Systems' Vulnerability Taxonomy	72
B	Event-Driven Systems' Threat Model	74
B.1	Base Test Environment Configuration	74
C	Opfuzz	78

List of Figures

2.1	Request-Response communication model	6
2.2	Event-Driven communication model	7
2.3	Event-Driven System, all nodes connected	8
2.4	Event-Driven System with a message broker	9
3.1	Publication Year distribution	20
5.1	Base experimental setup	40
B.1	Screenshot of example logs from policy authorizer	77
C.1	Screenshot of example validation report	78

List of Tables

3.1	Search Query Strings used	12
3.2	Article search results	12
3.3	Exclusion/Inclusion criteria	13

Listings

5.1	bad_schema.idl	41
5.2	Base environment changes for snappy-java experiment	42
5.3	Base environment changes for custom authorizer experiment	43
5.4	Auxiliary ACL tool	44
5.5	Base environment changes for ACL bypass	45
6.1	Example of payload that is missing some input or has incorrectly typed input . .	55
6.2	Example of payload that has values out of their accepted range	56
6.3	Internal topics may allow bypassing other restrictions	57
B.1	Base Experiment Environment	74
C.1	Example tool configuration	79

Abbreviations

ACL	Access Control List
DoS	Denial-of-Service
EDA	Event-Driven Architecture
EDS	Event-Driven System
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Manager
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
OPA	Open Policy Agent
RCE	Remote Code Execution
REST	Representational State Transfer
SDLC	Software Development Life-Cycle
WWW	World Wide Web

Chapter 1

Introduction

Contents

1.1	Context	1
1.2	Motivation and objectives	2
1.3	Document Structure	3

1.1 Context

Modern day cyber-ecosystems present a truly heterogeneous mesh of software services, ranging from the financial sector to academia and and going through government services. Almost everything we as humans do is centered around the digital world, with a DataReportal article from 2022 reporting more than double the number of Internet users since 2012, with social media user numbers tripling in that same amount of time [37]. Given all this usage, it is expected that as the number of normal every-day users grows, so does the number of cyber-threats. Recent studies have noted a steady but constant increase in threat escalation since 2010, hinting that this phenomenon is caused by the growing activity of cyber-criminals in the digital space, followed by cyber-warfare between nations trying to ascertain their virtual and political dominance over their adversaries.

In the presence of so many threats in cyber-space, it becomes crucial not just for individuals but for anyone who uses the internet either as a hobby, out of necessity or for professional purposes to be secured and protected. The first step to knowing how to prevent any threat from materializing is to know what it is trying to exploit, that is, what is its intended target. Even though different industries and economic sectors present very different requirements and specifications from one another, the one aspect they all have in common is that they all depended on Internet connectivity and, as such, rely on some form of mutually agreed communication paradigm. By far the most common type of communication in software systems is *synchronous communication*

wherein one party establishes some communication channel and waits for a response from the other party. This model has great similarities with human communication patterns, with a prime example being phone calls. However, these also evolve, and more sophisticated methods of communication emerge to face issues that others could not solve. As an example, if in a group of people everyone is waiting for someone else to reply to them, eventually the group and discussion will not move forward as everyone is infinitely waiting for someone who in turn is waiting for someone else, until eventually we would reach back the original person. Similarly, the same thing can happen with computer networks, where computers communicating between one another using synchronous communication methods could lead to degraded response times. The obvious solution is to take time completely out of the equation, meaning that all participants in the system communicate when they need to and might receive other forms of communication independent of when they was started or when they reached its destination. This form of communication can be called *asynchronous communication*, with a great example being postal networks.

Knowing what kind of systems we are working with not only reduces the scope of solutions one must need to work through but also allows for specialized knowledge to provide aid in preventing (or mitigating) future threats. However, to truly be able to secure a system and make it as future-proof as possible to potential attackers, it is important to know which problems persist that are specific for that kind of system. It is this need that powers any kind of security research and security testing into every type of software system: the better tested a system is, the more secure it should be.

1.2 Motivation and objectives

As software grows and systems become more complex, it becomes imperative that these same systems are tested against specific requirements that ensure the quality of the products being developed. Testing also ensures that potential threats are discovered early in the Software Development Life Cycle (SDLC), helping to prevent future vulnerabilities in these systems.

Testing synchronous web systems is relatively straightforward, as every request is either followed by a response or no response at all, which in itself is a valid result since it indicates a problem that the developers must attend to. There are multiple tools and frameworks for testing synchronous systems, one such example being Morest [43]. Furthermore, due to their simplicity, synchronous systems enjoy a lot more research than their asynchronous counterparts.

Testing asynchronous systems such as those modeled after Event-Driven Architectures (EDAs) becomes more complex due to the nature of the systems themselves: normal application workflows are comprised of sequences of events. Thoroughly testing these systems becomes infeasible due to the combinatorial explosion of event sequences that can (and must) be generated in order to test a system [45]. Furthermore, due to extreme decoupling, exercising one part of the system more often than not will be reflected elsewhere, making it also infeasible to track which events led to which changes in the system.

Nonetheless, efforts have been made towards testing event-driven systems [32], but as far as we could find none of those focus on systems specifically designed for and around the Kafka ecosystem. Furthermore, as far as we could find, no work has been done focusing on Apache Kafka threat analysis.

From these statements, we can see that there is a lack of evidences and understanding of what vulnerabilities and security flaws exist in Event-Driven Systems. As such we establish for this dissertation the goal of researching and studying threat analysis in the context of Event-Driven Systems and, more precisely, Apache Kafka. This research includes cataloging common vulnerabilities in Event-Driven Systems and assessing how they can be exploited in practice. From this, we can formulate a hypothesis central to this dissertation:

"By defining a taxonomy of vulnerabilities in Event-Driven Systems, (1) threat modeling for such systems can be simplified, and (2) the discovery of new vulnerabilities can be facilitated by providing a structured understanding of common attack vectors."

This kind of software analysis practices for asynchronous systems are desirable for companies that employ such systems, as is the case with Kuehne+Nagel¹, a global logistics company based in Switzerland that relies on event-driven technologies to drive its operations. This dissertation is thus developed in the context of and in collaboration with Kuehne+Nagel.

1.3 Document Structure

The document is organized in the following fashion:

- **Introduction** Chapter 1 lays out the context of this thesis regarding what technologies and methodologies exist, the problems with said technologies and methodologies and how those drive the motivation for the work done on this thesis.
- **Background** Chapter 2 introduces some concepts and terminology necessary to understand the contents of the rest of this dissertation and delves deeper into the concepts explained in Section 1.1.
- **State of the Art** Chapter 3 establishes the ground work for a Systematic Literature Review on Vulnerabilities in Kafka-based Event-Driven Systems and discusses the reviewed publications.
- **Problem Statement** Chapter 4 formalizes the problems and research gap that will be addressed by the work developed for this dissertation and formulates an hypothesis that will guide the remainder of the dissertation work.
- **Proposed Solution** Chapter 5 introduces a tripartite solution covering several aspects of the problems described in the previous chapter.

¹<https://home.kuehne-nagel.com/en/>

- **Validation** Chapter 6 establishes the methodology used to validate the solution against the hypothesis presented in Chapter 4.
- **Conclusions** Chapter 7 summarizes the content presented in this document and describes succinctly the work done. Contributions are listed, as well as limitations of the proposed solution components. Finally, points of improvement for future work are presented.

Chapter 2

Background

Contents

2.1 Request-Response Communication Model	5
2.2 Event-Driven Communication Model	6
2.3 Apache Kafka	9

In this chapter, some concepts deemed relevant to further clarify the motivation presented in the previous chapter and to contextualize terminology used throughout the rest of this dissertation document are (re-)introduced and explained. Before we talk about what Event-Driven Architectures and Event-Driven Systems are, a brief introduction to the traditional *Request-Response* architectural style should be given to contextualize the appearance of other architectural styles. This is done in Section 2.1. To contrast with the Request-Response model, in Section 2.2 *Event-Driven Architectures* are introduced. Finally, in Section 2.3 we introduce *Apache Kafka*, a distributed-log based streaming software product which empowers many of the concrete implementations of Event-Driven Architectures, along with a brief introduction of many of its flavors and distributions.

2.1 Request-Response Communication Model

Before understanding what Event-Driven Architectures are and how they came to be, we should first understand what existing technologies motivated their creation. In this sense, in this section we briefly introduce the concept of the *Request-Response* communication model.

Under this communication model, one participant in the message exchange, the requestor, sends a request to another participant, the replier, which processes the request, possibly performing some long-running operation, and then provides a response to the requester. This interaction happens synchronously, meaning that the requester is idle while waiting for the other party to provide a response. Message exchanges under this communication model can be simplified to the activity diagram present in Figure 2.1:

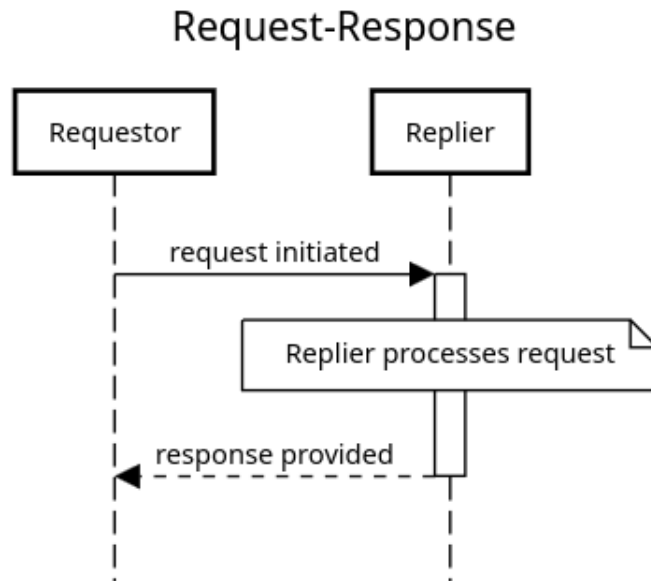


Figure 2.1: Request-Response communication model

Application protocols developed using this model include the FTP (File Transfer Protocol) and, more commonly known, HTTP (HyperText Transfer Protocol) [50], introduced by Berners-Lee and Fielding as a way to power the upcoming World Wide Web (WWW). HTTP has evolved to power most if not all of the software systems currently deployed on the internet in some way or another. One noteworthy innovation built on top of HTTP is the concept of *REpresentational State Transfer*, or REST, introduced by Fielding in his 2002 PhD thesis [26]. Even after two decades, REST continues to see major adoption, as stated in a 2022 online article about API Economy Statistics [66].

However, as the internet grew and software systems matured, there came a need for more immediate responses from system components. This need would later evolve into the development of *real-time systems*, which are capable of responding to events in near-real-time, a desirable characteristic in several economical and industrial areas. Due to its design, HTTP was no longer deemed the best fit for a communication protocol, which led to further research into novel communication models that could support the growing needs of the internet. One of the results of this research effort is the appearance of the *Event-Driven* architectural style

2.2 Event-Driven Communication Model

Under the Event-Driven communication model, exchanges between system components happen via events — significant/notable changes to an applications state [8]. Events may also be referred to as *messages*. There is not notion of different types of participants in an Event-Driven message exchange as there are in the Request-Response model. Rather, every participant can produce events and transmit them through the system — a so called *producer* application —, consume events, pulling a copy of them from the system and acting on their payload — a so called *consumer*

application — or they can act as both, possibly performing some kind of translation. Figure 2.2 illustrates a simple workflow in a system employing an Event-Driven communication model. To more easily talk about these kinds of systems, we will refer to them as *Event-Driven Systems*, or EDSs.

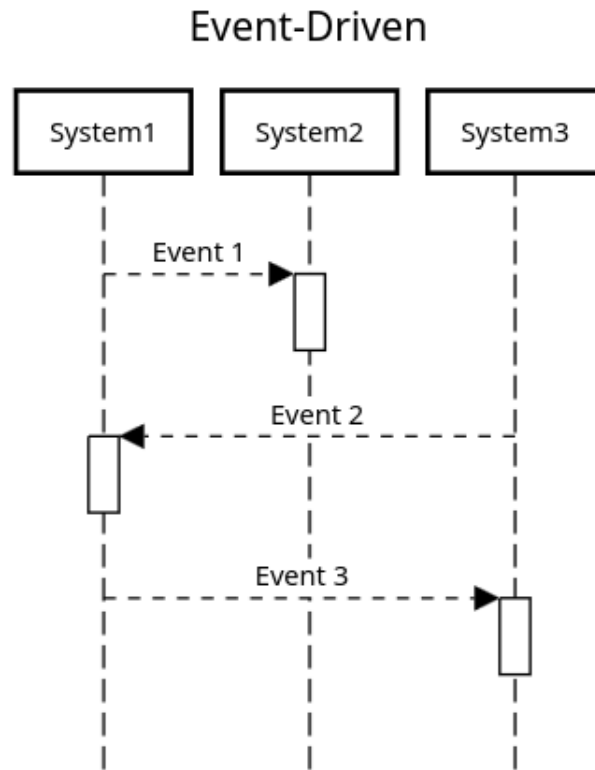


Figure 2.2: Event-Driven communication model

Using an Event-Driven System, different components can evolve at different rates, granted that they all share a common communication "language". This brings advantages when compared to the Request-Response model because it can improve developer efficiency, since teams only need to focus on their portion of the entire system. If some new business requirement needs to be implemented, instead of needing to possibly alter a massive pre-existing system and connecting all the dots, a new system component can be designed to solve that requirement and added to the Event-Driven System as a new participant.

Communication in an Event-Driven System might be based on HTTP exchanges between the various system components. However, this naive solution quickly becomes unscalable as more and more services are added, due to the inherent synchronous nature of the protocol itself, meaning that any latency in a message exchange will cascade to future exchanges. To solve these, protocols such as WebSockets have been designed to provide better efficiency in asynchronous communications.

The question remains on how producers and consumers can communicate with each other, that is, how do they know which participants are available. One solution could be to have every participant know about every other participant but, for obvious reasons, this solution would even-

tually not cope with the scale of the systems being operated, as the number of network connections would grow exponentially in the number of connected components. Furthermore, this approach would defeat the purpose of having separate producer and consumer applications since every system component would need to be aware of every other and, as such, would have to have domain knowledge they might not need. An example of this can be seen in Figure 2.3.

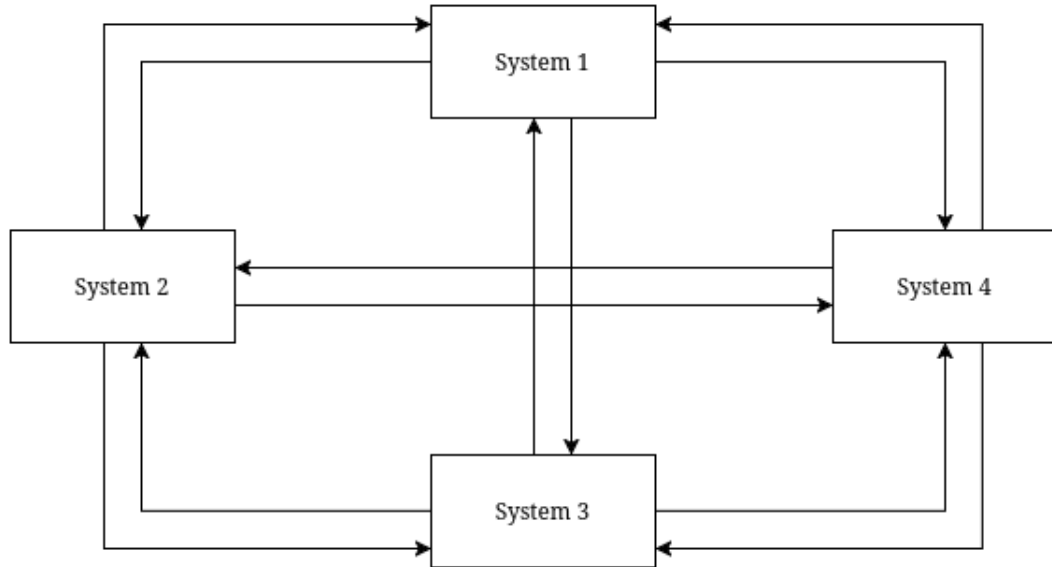


Figure 2.3: Event-Driven System where every node is connected to every other node

Another solution to this is the use of *message brokers*, software components whose sole responsibility is to broker the transmission of messages from one set of interested parties to another set of interested parties, independent of how many there are. This fits the event-driven nature of Event-Driven Systems since they can mediate message transmission from producers to consumers. One consequence of using message brokers is *extreme decoupling* between producers and consumers. Producers no longer need to manage or be aware of multiple consumers, and consumers need only to communicate with the broker. This intermediary role of the broker eliminates direct dependencies between producers and consumers, allowing each to evolve independently and scale without impacting the other, which is a desirable property of the system. Figure 2.4 exemplifies the role of message brokers in a system, where one can perceive immediately the reduced number of network connections in the entire system, which increases linearly with the number of components that are not the broker itself.

Due to the increased complexity of the system as a whole, Event-Driven Systems are more prone to suffer from issues and vulnerabilities that usually are not a concern for more traditional architectural styles. Problems might include insecure communication between components and incorrect state replication between components, among others. Usually, fixing these issues requires an extra degree of skill than it would for other types of systems. It is clear then that care should be taken to weight the consequences of implementing this type of systems against its benefits.

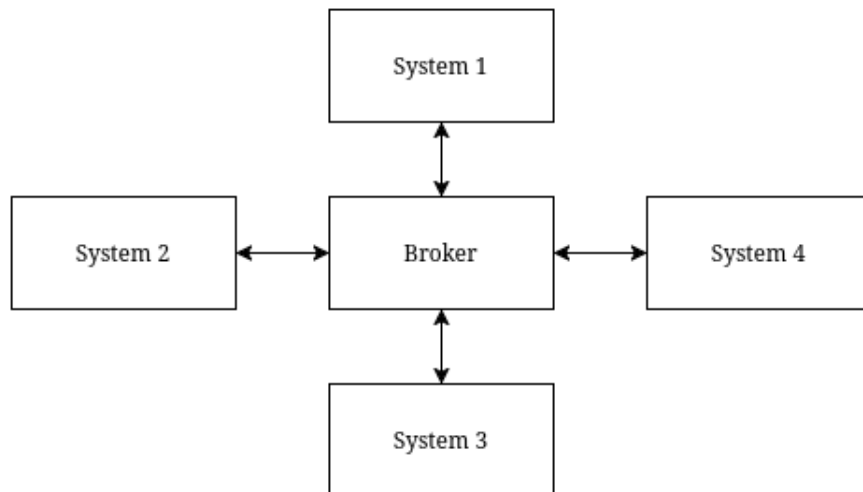


Figure 2.4: Event-Driven System with a message broker

There are other concerns that need to be taken into account when adopting this kind of component in an Event-Driven System, but it is clear how the role they play is essential to the scalability and performance needed for operating modern applications and software systems, and as such message brokers have been a widely developed technology, with examples such as **RabbitMQ** [7] or **Apache Kafka** [27].

2.3 Apache Kafka

Apache Kafka, originally developed at LinkedIn "for collecting and delivering high volumes of log data with low latency" [39], is an open-source, real-time, distributed, event streaming platform, built using a *distributed log* architecture and powering some of the leading tech companies in the world [28]. Its communication model is based on a *publish/subscribe* (pub/sub) model: *producer* applications are those that produce messages and publish them to Kafka, while *consumer* applications subscribe to those same messages, fetch them from Kafka and process them. As described in the previous section, client applications can serve as both a producer and a consumer.

The main primitive of Apache Kafka is the *topic*, which can be thought of as a queue of events/messages ordered by publish time. Topics can have zero, one or multiple producers publishing messages to them. Topics can have zero, one or multiple consumers subscribing to them and consuming messages stored in them. Each consumer is responsible for keeping track of their offset in a topic. By advancing this offset, consumers are fed with messages as they are published to the topic. Topics are distributed between multiple *partitions*, with each partition being allocated to a separate cluster node. Message ordering is guaranteed for messages inside the same partition, but global ordering is not guaranteed. Messages with the same key will always be stored on the same partition. Partitions are further divided into *log segments*, which are the physical files storing log data. Only one segment is in the `ACTIVE` state at every point in time. Once the maximum size in bytes or the time limit for events in that segment is reached, Kafka automatically closes the

current segment and opens a new one. Kafka's retention policies control the lifespan of individual segments. All of these mechanisms work together to ensure that Kafka is a highly distributed, highly scalable application capable of a high number of concurrent connections.

When a consumer application is connecting to Kafka, it needs to specify its *consumer group*. A consumer group is a logical grouping of consumer applications that, from the cluster's perspective, is the actual consumer fetching records. Consumer groups keep their own offset for each topic. Consumer groups are useful in scaling parallel stream processing in Apache Kafka by making it so that each node in a group is responsible for processing messages fetched from a different partition. This assures that each partition is assigned to only one node and as such no data conflicts can occur from multiple consumers working on the same data at the same time. Furthermore, fetch requests made by consumers in a group are load balanced between those same consumers by the group when it fetches logs and offsets from the main cluster.

There are many distributions of Kafka, such as those provided by major cloud providers like Amazon MSK, Google Cloud Managed Service for Apache Kafka, Confluent Cloud or the Confluent Platform [4, 12, 13, 31]. By far the most used of these is Confluent Platform, offering extra features like a Schema Registry for storing Avro schemas and ensuring messages in a topic comply with said schema, ksqlDB (a stream processing SQL engine) and Kafka Connector, a separate service that can be deployed alongside Kafka and allows ingestion from data sources like databases or delivery to different data sinks such as document indexes.

Chapter 3

State of the Art

Contents

3.1	Methodology	11
3.1.1	Databases	12
3.1.2	Publication search	12
3.1.3	Filtering	12
3.2	Results	13
3.3	Discussion	18
3.4	Evolutionary Analysis	20
3.5	Summary	20

To assess the contributions already made on this thesis' subject present in the literature and/or proposed by the industry, a Literature Review was conducted. As a first step, only records from the academic literature were considered, in order to assess the development level of academia research on this topic.

This chapter is organized as follows: in Section 3.1 we present the methodology used to conduct this review, presenting the results obtained from that in Section 3.2. A short summary of the results is presented in 3.3 and threats to the validity of this literature review and final remarks are given in Section 3.5.

3.1 Methodology

In this section, the methodology employed to conduct the literature review is outlined. In Section 3.1.1 we present the bibliography databases used to query for literature results. Next, Section 3.1.2 details the query strings used to query the aforementioned bibliography databases to retrieve publication candidates for analysis. Section 3.1.3 describes the step to filter the previous results

down to a set of articles which not only are more manageable but also are more relevant to the research at hand.

3.1.1 Databases

In order to correctly assess the current situation in the field being studied, high-quality and trustworthy sources of information must be analyzed. As such, article discovery relied on famous and reputable bibliography databases such as IEEE Xplore, ACM Digital Library and Scopus.

3.1.2 Publication search

In order to capture the greatest amount of relevant publications pertaining to the topic of automated penetration testing of Kafka-based systems using side-channel information, two queries were created:

QS1	(kafka OR Kafka) AND (threat OR vulnerability)
QS2	((async* AND (system OR architecture)) OR (event?driven AND (system OR architecture))) AND "penetration testing"

Table 3.1: Search Query Strings used

This initial search process resulted in 769 candidate publications prior to filtering. The results are summarized in the following table:

Library	QS1	QS2	Total
ACM DL	527	138	665
Scopus	84	5	89
IEEE Xplore	14	1	15
Total	625	144	769

Table 3.2: Article search results

3.1.3 Filtering

To obtain the relevant articles from the collected list, the following **Exclusion Criteria** and **Inclusion Criteria** were applied:

After applying the exclusion criteria, there were 401 publications remaining. Of these, the title and abstract were analyzed according to the inclusion criteria defined to find which are suitable for a more comprehensive analysis. This yields a total of 17 publications, which are further analyzed in the next section.

Identifier	Description
EC1	The publication is not written in English
EC2	The publication dates back to before 2018
EC3	The publication is a duplicate
EC4	The publication is a Book, Book section or Magazine publication
IC1	The publication covers topics about asynchronous, event-driven systems/architectures, with a special focus on Apache Kafka
IC2	The publication covers topics about threat/vulnerability detection and system testing

Table 3.3: Exclusion/Inclusion criteria

3.2 Results

Datta et al. [21] introduce *Valve* as a serverless compute platform built around full transparency of the operations of information flows built on it. It does so by introducing a specialized *agent* in each "function-instance" (i.e., container, which is used interchangeably), responsible for monitoring file and network I/O of the instance itself and reporting those to a *centralized controller* to determine the information flows of a serverless application as a whole. The controller is responsible for deriving a permissive security policy, which is (1) uploaded to all agents to enforce information flow policies; and (2) shared with workflow developers to further refine the policies and secure systems. The policies themselves are specified in a DSL, developed for this purpose. Main design goals include being platform and language agnostic while being sufficiently expressive to aid workflow developers.

Testing results show that while *Valve* introduces operational overhead when compared to a vanilla deployment of OpenFaas this is negligible (around 2.8%). Furthermore, the authors show that 2 common attack patterns in serverless systems are mitigated by the security design of *Valve*.

Ilca et al. [42] propose a method for security analysis of Windows Communication Foundation (WCF) systems using offensive measures. The existing literature related to WCF security does not focus on offensive measures, which are as important, if not more, than other security measures, partially resulting from the expansion and growth of WCF into other platforms and systems. Having set an "ABC" definition — Addresses, Bindings, and Contracts — paves the way to understand how each component in a WCF system might guarantee secure communications. Despite all of these features, there are still actionable exploits present which need to be addressed.

A step-by-step methodology for offensive analysis is presented, starting from analysis of publicly disclosed *Web Services Description Language* (WSDL) files that could hint and misconfigured services or exposed credentials. These files and insecure service endpoints can be obtained using a technique called "Google Dorking". Then, these insecure WCF endpoints can be explored and analyzed using automated tools, built either for generic SOAP/REST APIs or specifically for WCF. Examples of possible vulnerabilities are reverse shells. Afterwards, vulnerability analysis tools can be used to understand exactly where vulnerabilities might appear and how they impact

affected services. Finally, other specialized tools might be used to assess the level of security of the transport layer in these WCF services.

Kreyssig and Bartel [40] analyze several combinations of JDK releases with common Java dependency libraries to discover which ones enable the use of gadget chains for deserialization vulnerabilities, extending previous work. Gadget chains in the context of deserialization are similar to gadget chains in Return-Oriented Programming (ROP). They find that the majority of exploits result in Remote Code Execution and that most vulnerabilities are not disclosed by CVEs and the existing disclosure processes, hinting at further developments in this area. Furthermore, it is shown that recent versions of all the JDKs, dependency libraries, and analyzed projects contain these gadget chains, demonstrating a lack of security awareness towards deserialization vulnerabilities. Many of the exploits considered work for major open source projects such as Apache Kafka.

In light of this, a tool, *Gadgacy*, was developed in order to assess whether any given combination of runtime environment and library dependencies could enable the use of the aforementioned gadget chains. It is shown, using this tool, that some of the exploits are present in a vast majority of the analyzed projects, further validating the obtained results.

Srivastava et al.[68] propose *Crystallizer* as a novel, state-of-the-art framework for detecting deserialization vulnerabilities in Java applications using both static and dynamic analysis. Library and application code is statically analyzed, producing a call-graph from which a *gadget-graph*, a subset of the call-graph composed of gadget chains, can be obtained. Using this graph, dynamic methods allow Crystallizer to probabilistically synthesize payloads that exploit gadget chains in the analyzed applications.

Results include a large improvement in the number of gadget chains and gadgets found for libraries which have been analyzed as part of previous research. Furthermore, to showcase the utility of the tool, the authors used it to uncover previously known but also unknown gadget chains in famous open-source projects such as Apache Kafka and Apache Pulsar.

Baban [5] proposes a novel technique for data pre-processing, with a focus on data validation, in the area of Big Data applied to energy systems using industry standard Stream Processing Engines (SPEs) like Apache Flink or Kafka Streams. These SPEs support batch and real-time processing, and the latter is more suitable for the context of energy systems. However, they do not support data validation before-hand out-of-the-box, motivating the work and the development of a system that does so before passing the data to the SPE itself.

Due to its low latency, Apache Flink is used as the validation system that feeds into the actual SPE, in this case Kafka Streams, which was chosen due to its resilience and fault-tolerant features. Future work is aimed at expanding the work and eventually distribute it as a library for general use.

Zdun et al. [71] developed a novel approach for evaluation of microservices architectures according to *Architectural Design Decisions* (ADDs). The design decisions were developed with a focus on observability, identity and secure communication. 10 case studies were performed based on models of microservice systems in order to test several variations of various components and added features. For these, a total of 4 ADDs were defined, and a 5-point scale was used between each pair of case-study and ADD.

Shahoor et al. [63] developed and presented a technique to automatically fix bugs present in reactive programs, namely those having to do with blocking function calls. They mined Github for famous open-source projects that employed commonly used reactive Java frameworks (Reactor, RxJava, etc.) and Github issues/Stack Overflow discussions for potential fixes to those same issues found. Patches for the issues found were created and applied. Finally, GitHub pull requests were created for the repositories studied, possibly being accompanied by a performance evaluation report (i.e., whether the bug-fix improves performance or not).

Testing shows that: (1) blocking-call bugs are the most prevalent bugs out of all *reactiveness bugs*; (2) the bug-fix patches generally improved performance (with some outliers that can be ignored when all recorded metrics are taken into account); (3) developers tend to accept changes to their code-bases if there exists evidence of the improvements that change may bring.

Zhang et al. [72] applied a white-box fuzzing tool (EvoMaster) to 40 APIs of a company in order to assess how the use of automated tools could benefit industry practitioners and developers. EvoMaster is an open-source tool that enables automated testing of systems through fuzzing. It supports *white-box* and *black-box* testing for several languages and API technologies, such as REST, RPC or GraphQL. EvoMaster itself was modified to support (1) an extensible and technology agnostic abstraction of external services and databases; and (2) test seeding as a means to make test generation and test case evolution not only faster but more precise as it is based on actual data.

After modifying the tool to support service mocking and test seeding, they found that the extended version of the tool generally provided better results than both the in-house testing and the base version of EvoMaster in terms of *Line Coverage* and fault detection. Furthermore, two time budgets were used (1 hour and 10 hours), with the largest budget consistently providing better results on all configurations of the tool.

Duarte et al. [24] study how to exercise fault-tolerance mechanisms in Internet of Things (IoT) via deliberate fault injection. The importance of fault-tolerance in software systems, especially in IoT contexts, directly motivated the topic of this research. An existing MQTT broker was instrumented to allow configuring fault-injection behavior directly in the broker, without relying on external components that would connect to the system. Node-RED is used as the programming environment as it provides a low-code, visual system development environment. Previous work on this field also motivated the use of Node-RED.

Experiments on a real-world dataset revealed that fault-injection is an important test avenue for assessing the quality of IoT systems, especially the ones using self-healing mechanisms. When considering all the variations of the system under test, the systems employing self-healing capabilities consistently performed better than the ones who didn't, even in the presence of deliberately injected faults.

Waseem et al. [69] performed an empirical study where they analyzed over 1000 issue discussions on GitHub related to 5 major open-source micro-services systems in order to discover the largest types of problems faced and what triggers them. This led to the creation of a "first of its kind taxonomy of the types of issues in open source micro-services systems". In this study, 17 main categories of issues were discovered and promptly organized, although only the 4 biggest ones (technical Debt, Build Issues, Security Issues, Service Execution and Communication) were covered, and for each a succinct explanation of their sub-categories was given.

Allied to these, the authors presented a list of the major causes related to the previously mentioned issues, the main one being General Programming Errors. This study serves as a great starting point for micro-services practitioners to improve their code-bases and systems.

Das et al. [20] present *HawkEDA* as a solution to data anomaly management in micro-services architectures. The architectural design of HawkEDA comprises several components that allow developers to specify precisely the events their applications produce and consume and expected invariants of application workflows, such as event precedence. It also has the ability to connect to common message broker technologies in order to retrieve messages for analysis.

When applied to a sample Github repository ¹, results show specific configurations of the tool which minimize anomalies, attesting to the tool's usability, even though it is still not suited for the general public due to the technicalities needed to implement it.

De Bleser et al. [22] present a novel approach to test distributed actor-based Scala applications built with the Akka framework for resilience issues, namely message duplication and actor restarts. This approach is materialized in a tool, Chaokka, which, for the Scala programming language, instruments a target application and test specification and attempts to determine which perturbations affect the test outcomes, which directly relates to the resilience of the system under test. The tool uses *delta-debugging* to explore the state space of possible perturbations, leading to more performant verifications.

Testing has showed that the use of a delta-debugging approach can increase up to 4 times the performance of resilience testing when compared to a random approach. Further optimizations were done but these are highly dependent on the system under test and, in the worst case, degrade to de unoptimized delta-debugging approach.

¹<https://github.com/dotnet-architecture/eShopOnContainers>

Zhou et al. [73] developed *MEPFL*, a microservices error-prediction and fault-localizer tool that leverages information from system trace logs. After automatically extracting a set of features that log traces contain, the tool trains prediction models on both an healthy version of the system under test and a faulty version, instrumented through fault-injection. Three microservices fault types are supported: multi-instance, configuration and asynchronous interaction faults.

After being experimented on two open-source microservices (one of them being the biggest example to the best of the authors' knowledge), it is shown that the tool achieves high recall/precision with a low rate of false positives.

Khatuna Jannat et al. [38] introduce a new technique to analyze real-time internet traffic to detect ransomware attacks via deploying the Advanced Encryption Standard (AES) on Kafka Streams. Data is collected and filtered by the Kafka Streams platform and then encrypted using AES, thus ensuring compliance with existing data privacy standards. After being disseminated through the correct topics, a machine learning model is used and trained on 70% of data in order to correctly label possible traffic as being part of a ransomware attack (with label "1") or not ("0").

Experimentation results showed an average accuracy of around 97%, highlighting not only the usefulness but also the reliability of this technique. Furthermore, the use of Kafka Streams as the underlying backbone of this setup is motivated by its speed and performance.

Kaushik et al. [36] explore the *Log4Shell* vulnerability found in the open-source *Apache log4j* project, presenting a method to explore and exploit vulnerable systems. The 7-step method is based on an online learning environment ² for exploring and enacting an exploit based on Log4Shell. The exploit includes Reconnaissance, Information Gathering, Vulnerability Exploitation, Access Maintenance, Detection, Security Bypassing and Mitigation.

Singh and Chaudhary [67] describe a new framework for securing communications with event-bus backends, called *REST Security Framework* (RSF). This system arises from the need of securing communications and operations on common event-streaming platforms by client applications, such as RabbitMQ and Apache Kafka which offer their own security features. These security features are not, however, enabled by default or the default configurations are insecure, leading to further work needed to be done. The proposed RSF is comprised of 2 sub-systems: *Identity Access Control* (IAC) and *Security Gateway* (SG). The IAC is responsible for user and application registration, token distribution and identity verification. The SG is charged with mediating communications between a client application and an event streaming platform in a secure fashion. The authors also delineated a Threat Model for their RSF, outlining potential threats and the mitigation measures taken to prevent them.

²<https://tryhackme.com/room/solar>

Testing results when using Apache Kafka as the underlying event streaming platform show that, compared to other mainstream libraries, like `python-kafka`, RSF enjoys a significant performance boost when handling a high number of concurrent users, attesting to the tools reliability and security.

Huang et al. [33] introduce *BPS*, a blockchain-based publish/subscribe platform built for security in multi-tenant environments. BPS leverages blockchain technology to store critical metadata used to ensure integrity of the operations performed (i.e., only authorized tenants can perform operations on topics/brokers/messages they are authorized for). The proposed prototype is built on top of the EoS blockchain and Kafka pub/sub technologies. Alongside the blockchain features, a reward system is set in place that aims to deter and ultimately stop malicious actors from disrupting the system. Even so, every action in the system is logged, so accountability can be ascertained. A hybrid-storage methodology is used to cope with the large-scale traffic expected from the use of the tool.

Experiments show that the blockchain validation features of BPS introduce only a small overhead when compared to regular Kafka for the same amount of message traffic, converging as this number increases due to caching of access requests.

3.3 Discussion

The analyzed articles cover a broad spectrum of research interests. That said, none of them closely approach the research topic of this thesis. This points to a clear research gap within the literature, limiting the possibility of expanding or adopting previous research directly on this thesis but providing a great opportunity for contributions regarding security of Kafka-based Event-Driven Systems. One possible way to mitigate the lack of background-research is to expand the analysis done in this chapter to *grey literature*, since those are usually produced by industry practitioners and the theme of this thesis is industry-related. Nonetheless, the reviewed articles still provide valuable insights which might be useful for further work and so they are promptly discussed.

Deserialization issues in Java applications were brought up by 2 articles [40, 68]. Although these articles do not directly mention EDAs, they refer to Kafka as one of the example projects for which they tested their hypotheses, and with Kafka being at the core of many EDAs, it is valid to extend those hypotheses to not only full Kafka-based EDAs but potentially other event-driven ecosystems. Further uses of Kafka include a blockchain-secured pub/sub platform [33] that provides security guarantees in multi-tenancy environments, a REST-based security framework for event-streaming platforms which protects the underlying message broker from unauthenticated and unauthorized accesses [67], a novel use of AES on the Kafka Streams platform for real-time traffic analysis to detect ransomware attacks [38] and lastly a Big Data pre-processing platform making use of the distributed and decentralized nature of Apache Kafka [5]. Even though [33] and [67] develop security features in systems using Kafka, their work does not focus on security of the Kafka platform itself or of systems deploying a vanilla or commercial version of Kafka.

[36] explored a famous vulnerability found in a ubiquitous Java package (which itself is used by almost, if not all, of the Apache Ecosystem projects, including Kafka), although not thoroughly.

On the topic of anomaly and fault detection, [20] present HawkEDA as a solution to manage operations in EDAs according to their data-flow and take action accordingly. [22] provide a tool which leverages delta-debugging to uncover resilience issues in actor-based Scala programs, which can be directly "translated" into microservices systems; and [73] developed a tool for latent error-prediction and fault-localization in microservices through the use of prediction models. Microservices architectures more often than not are implemented on top of a EDA and, as such, can be used to test and assert certain properties of these systems, with Kafka being an example of being one such system. [72] explore and extend EvoMaster on 40 RPC APIs of a company to explore how the tool's white-box testing and fuzzing capabilities, paired with the aforementioned extensions, can help bring up more issues with code-bases and run-time behavior. RPC APIs are also used in conjunction with EDAs and so the extended EvoMaster might be a suitable choice for testing these kinds of systems.

Valve [21], as a full-blown FaaS platform, provides its users with minimal overhead for more restrictive security policies, preventing certain kinds of exploits on any existing vulnerabilities in the deployed systems. [24] study how instrumenting an MQTT broker with developer-controlled fault-injections might influence the evolution of a system's state and behavior. Knowing how fault-injection mechanisms and security features of serverless platforms are implemented can prove crucial in understanding what issues might be present in common EDAs and how to discover them.

Still on asynchronous and event-driven systems, [42] propose a method for offensive security analysis of systems employing the Windows Communication Foundation (WCF) framework by presenting a step-by-step guide on how to exploit these systems. While this article is specific to a single technology stack and, moreover, does not concern Kafka, the offensive security analysis methodology can be adapted and, as such, deployed on other event-driven ecosystems.

A taxonomy of known issues in microservices architectures was developed in [69]. This taxonomy includes categories such as Build Issues and Technical Debt, which are present in most if not all considerable software projects. While this categories were defined for the analyzed microservices projects, we can extrapolate some of the defined categories and sub-categories into the use of EDAs and, specifically, Kafka-based systems.

In order to understand the current problems faced by reactive programs and the reactive programming paradigm as a whole, [63] presents a technique for uncovering and mitigating blocking-call-related bugs in code-bases using reactive-frameworks, such as Reactor for the Java ecosystem. Since EDAs fall into the category of reactive systems, these are also susceptible to blocking-call bugs unless correctly engineered, making this a valid point for research.

Finally, [71] present a microservices evaluation approach, based on statistical models and automatic metric extraction, to assert three aspects of the microservices systems under test. Once again, EDAs can serve as the backbone for many microservices systems, which need to be correctly protected and secured against malicious actors/threats.

3.4 Evolutionary Analysis

Given the previously defined set of Inclusion and Exclusion Criteria, namely EC3, it becomes obvious that the analyzed works were all published since 2018, and as such an evolutionary analysis can only take into account a fairly limited amount of data. Nonetheless, in Figure 3.1 we present a graph showing the evolution of released articles and publications on the topic under research.

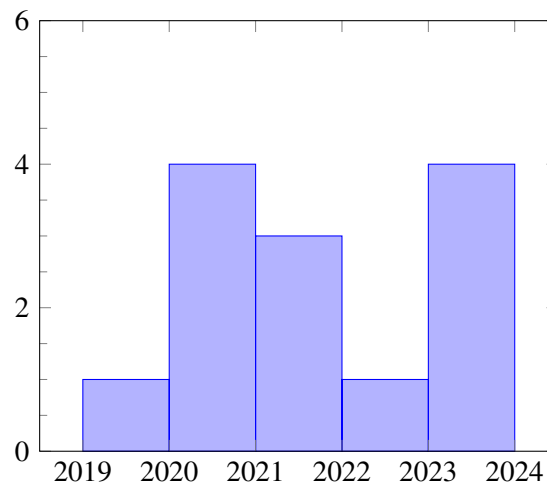


Figure 3.1: Publication Year distribution

No trends in the evolution of academic research regarding vulnerabilities in EDAs can be seen from this data, possibly hinting at a research gap in the context of EDA vulnerabilities in Kafka-based systems.

3.5 Summary

A literature review was done on the topics of vulnerabilities in EDAs and namely Apache Kafka. Out of the 769 initial publication candidates, only 17 articles remain after filtering. These were then read, analyzed and discussed to get a grasp on the state-of-the-art of EDA vulnerability research. Beyond inherent bias in the interpretation of the contents of the publications against the exclusion and inclusion criteria and lack of refinement of the search query strings, no further threats to the validity of these results could be identified.

To the best of our knowledge, there hasn't been much research done with regards to vulnerability testing in event-driven architectures specifically. This shows that there is a research gap in the literature that can and should be explored. Nonetheless, various efforts have been made in indirectly related areas.

Further analysis could expand on the work done in this review by also leveraging sources from *grey literature*, as these better reflect the development efforts made by industry practitioners. As this thesis' theme is more connected to industry efforts, those could provide a valuable source of information.

Chapter 4

Problem Statement

Contents

4.1	Problem	21
4.2	Hypothesis	22
4.3	Research Questions	23
4.4	Summary	23

Following the work done in Chapter 3, in this chapter we aim to provide a formalization of the current problems in vulnerability categorization for Event-Driven systems and what work is planned to solve those problems.

In Section 4.1, a description of the problems with vulnerability research in Event-Driven Architectures and Kafka-based systems is formalized. Based on those, a hypothesis is formulated in Section 4.2, from which the rest of the dissertation work is developed. Finally, a summary of all the previous sections is presented in Section 4.4.

4.1 Problem

Security in software systems and software development has always been a major concern, with great care taken to ensure that developed products and systems are as resilient as possible to external threats. This care takes shape in very different forms depending on the existing kinds of system: a monolithic, non-networked application will experience a significantly different threat surface from a distributed, multi-service application running across several different networked locations. Many efforts exist to categorize security risks present in software, chief among them being the *OWASP Top Ten* [55], a list of common web application security risks, presented as "*... the first step towards more secure coding*" and compiled from security accounts from third-party entities. Examples of categories taken from this list include *Broken Access Control*, *Software and Data Integrity Failures* and *Security Misconfiguration*. However, this list covers the landscape of

web application security risks in a broad fashion, cataloging general risks (and concrete examples of these risks) without associating them with a particular type of system from which those same risks arise.

More focused to a particular area is the *OWASP Cloud-Native Application Security Top 10* [53], detailing common security pitfalls experienced by developers when building Cloud-Native applications, often implemented as part of an Event-Driven System. Some of these pitfalls include *Insecure cloud, container or orchestration configuration, Improper authentication and authorization* and *Inadequate ‘compute’ resource quota limits*, just to name a few. However, this list is outdated, having been last updated in 2022, meaning that some of the insights it aims to provide might be outdated due to the rapid evolving nature of the Cloud-Native ecosystem. Furthermore, accessing the links available on the OWASP Cloud-Native Application Security Top 10 official document website leads to a Github repository¹ where work was initiated to construct an extensive and detailed account of Cloud-Native security risks but was eventually abandoned (as can be seen by the lack of frequent updates, i.e. commits, to the document and its components).

It is clear then, from both the reviewed literature in Chapter 3 and the existing documents available, that there is a clear gap in research focused on vulnerabilities and security concerns in Event-Driven Systems. This can have 2 possible reasons to be: either there’s a lack of interest from both the academia (as can be seen in Figure 3.1) and industry, or industry practitioners actively research EDA vulnerabilities but, since these are scoped for their own systems, often times containing sensitive or even confidential information, these findings remain unpublished to the general public. It should also be noted that performing any form of security audit in this kind of systems is a harder task when compared to other, simpler systems design-wise, due to the fact that interactions in an Event-Driven System follow a different model that requires not only more mental effort but also more infrastructure and system support to process and arrange into a traceable audit that can lead from a known vulnerability to its source and mitigation efforts.

4.2 Hypothesis

From the problems defined in Section 4.1, the work developed for and presented in this dissertation aims to validate the following hypothesis, first introduced in Chapter 1:

"By defining a taxonomy of vulnerabilities in Event-Driven Systems, (1) threat modeling for such systems can be simplified, and (2) the discovery of new vulnerabilities can be facilitated by providing a structured understanding of common attack vectors."

Here, *simplified* means that developing such a threat model is made simple due to it requiring *ad-hoc* efforts and, as such, does not suffer from the complexity of more established methodologies; *facilitated* means that understanding common attack vectors is paramount to understanding how to protect systems against those same vulnerabilities.

¹<https://github.com/OWASP/www-project-cloud-native-application-security-top-10>

4.3 Research Questions

Based on the aforementioned hypothesis, we can synthesize the following Research Questions:

RQ1 What are the main security vulnerabilities present in systems employing an EDA and, more specifically, Apache Kafka?

RQ2 What mechanisms exist for malicious threat actors to exploit those vulnerabilities against arbitrary systems?

RQ3 How can we provide a structured view of common attack vectors plaguing Event-Driven Systems?

4.4 Summary

As seen by the Literature Review conducted throughout Chapter 3, there's a lack of adequate research available regarding vulnerabilities in Event-Driven Systems, either due to a lack of general interest from researchers or due to privacy concerns with affected systems. As such, we hypothesize that *given a taxonomy of vulnerabilities in Event-Driven Systems and a Threat Model that helps exercise those vulnerabilities, a tool designed around these two pieces can aid in uncovering previously unknown vulnerabilities in Event-Driven Systems*, which will be validated with the help of industry experts, capable of correctly assessing the results of this tool.

Chapter 5

Proposed Solution

Contents

5.1	EDA Vulnerability Taxonomy	25
5.1.1	Context	25
5.1.2	Methodology	26
5.1.3	Results	27
5.1.4	Categorization	34
5.1.5	Threats to Validity	38
5.1.6	Summary	38
5.2	EDA Vulnerability Threat Model	39
5.2.1	Context	39
5.2.2	Test Environment	40
5.2.3	Summary	45
5.3	Policy Vulnerability Detection Tooling	46
5.3.1	Rationale	46
5.3.2	System Components	48
5.3.3	Summary	49
5.4	Summary	51

Attempting to validate the hypothesis formulated in the previous Chapter (4, Section 4.2), a tripartite solution was designed: first, a number of articles from the gray literature were collected, analyzed and compiled in order to create a *Categorization of Event-Driven Vulnerabilities*, described in Section 5.1. Based on the categories found in the previous step, a practical exploration of those vulnerabilities is performed, and the results are gathered to form a *Threat Model* of Event-Driven vulnerabilities, detailed in Section 5.2. Finally, in Section 5.3 and based on the developed threat model and using insights from the constructed taxonomy of Event-Driven vulnerabilities,

we introduce *Opfuzz*, a Proof-of-Concept tool capable of stressing one particular aspect of Event-Driven Systems: authorization via policy enforcement. Finally, a concise summary of the contents of this chapter is given in Section 5.4.

5.1 EDA Vulnerability Taxonomy

5.1.1 Context

As was detailed in Section 4.1, software security is paramount to the correct operation of any kind of system, and should thus be treated as a major concern when designing and building those same systems. This is made all the more clear once we take into account reports of the growing threat to software systems, as was the case in the UK in 2024 [61]. There are multiple efforts to bring software security to the forefront of the SDLC and software operations. A major step in improving a system's security posture is through early identification and diagnosis. One example of this lies in the development and proliferation of **Common Weakness Enumerations**(CWEs)[51] and **Common Vulnerabilities and Exposures**(CVEs)[19]. These aim to

"provide(s) a common language of discourse for discussing, finding and dealing with the causes of software security vulnerabilities as they are found in code, design, or system architecture"

and

"identify, define, and catalog publicly disclosed cybersecurity vulnerabilities"

, respectively. Identification of a system's vulnerabilities and weak points implies some sort of work going into understanding that system's threat landscape. Still, these only reference specific, particular instances of vulnerabilities and weaknesses. This can prove sufficient at an organizational level, where development efforts are solely focused on the organization's systems or software. Furthermore, many of the identified CVEs relate to software that is free for use, i.e., open-source, and as such every client or dependent that uses an affected piece of software gets the benefits of being informed of potential security risks in their software and if and how those risks can be mitigated. However, their specific nature means that these mitigations needs a particular environment to be applied and effective, which may not always be guaranteed.

On a broader, more encompassing scale, there are the OWASP Foundation's Top 10 projects, chief among them the homonymous OWASP Top 10 ranking [55]. These serve as a baseline for software organizations and individuals to ensure

"that their web applications minimize these risks"

and

"OWASP Top 10 is perhaps the most effective first step towards changing the software development culture ... into one that produces more secure code".

These rankings are formulated from a set of donated security reports, identifying systems under test, CWEs that were tested, the type of tests performed, among others. Since they encompass a larger portion of the software ecosystem, a lot of the recommendations there can be applied to several, seemingly unrelated systems, as long as they share some connection in their design and architecture, be it a common serialization format, a common authentication and authorization middleware, etc. Examples of other, more domain specific OWASP rankings include: the OWASP Cloud-Native Application Security Top 10 [53], the OWASP Mobile Top 10 [57] and the API Security Top 10 [56], to name a few.

A quick internet search leads to the conclusion that there is no domain-specific ranking for Event-Driven Systems counted among the OWASP Foundation's projects. Furthermore, there seems to be no standard ranking of vulnerabilities in Event-Driven Systems like there are for other ecosystems, as was seen previously. The closest alternative is the OWASP Cloud-Native Application Security Top 10, since many Cloud-Native applications are implemented using an Event-Driven Architecture. However, this category attempts to fit the entirety of the Cloud-Native system, thus including references and findings that cannot be applied to the context of Event-Driven Systems. Furthermore, exploring the OWASP Cloud-Native Application Security Top 10 quickly uncovers that this project, although it is advertised as being under development and "very much alive", hasn't seen significant updates since July 2022.

As such, through the work done in this dissertation, we set out to create a first-of-its-kind taxonomy for Event-Driven Systems, collected from reports and accounts gathered in the gray literature. This taxonomy should list all discovered categories, along with examples of how a vulnerability falling in a given category can not only be exploited but also mitigated.

5.1.2 Methodology

In order to correctly understand the threat landscape surrounding Event-Driven Systems, we must take into account results gathered by industry practitioners from the gray literature. Although this method is not as rigorous as the one used in Chapter 3, a form of methodical process should still be applied. After carefully reading and analyzing each article, their main takeaways, as well as some other pieces of information, are registered in a custom spreadsheet whose goal is to provide a structured base from which conclusions about the taxonomy can be drawn.

Query Definition Before exploring any articles found through web searches, it is important to define *what* should be looked for. Common keywords related to *Event-Driven Systems* and *vulnerabilities* include "Confluent", "Policy mismatch" and "serialization". A more extensive list of search keywords can be found in Appendix A.

Article Discovery The query strings defined in the previous step were fed to famous search engines, such as Google, and the results present on the first page of search results collected for further discovery. Many of the results found, despite coming from different websites and different sources, mention the exact same vulnerabilities, CVEs or security risks, and so they were excluded.

Article Analysis Each article or blog post was carefully analyzed according to 4 main dimensions:

- **How** indicates what exact action needs to be performed to trigger the vulnerability;
- **When** describes the order of steps that need to occur up until (and including) the action defined in "How" for the vulnerability to trigger;
- **Where** indicates in which environments the analyzed vulnerabilities can occur;
- **Why** indicates possible other issues with the existing system or some of its dependencies that led to the appearance of this vulnerability;

Other relevant data that was collected includes *affected software versions*, *prevention methods* and a link to the article under analysis. Notes were also taken, summarizing the findings of each article or providing further context not encoded by the aforementioned dimensions.

It is worth noting that some of the reviewed articles did not provide answers to some or all of these categories. In such cases, a value of *N/A* was used, signaling that the data for that dimension was *not available*. Another aspect of this methodology that is important to consider is that, more often than not, the data present in all categories overlaps in some way or another, since they effectively are derived from the interpretation of the article itself. This obviously can threaten the validity of these findings and is briefly discussed in Section 5.1.5.

5.1.3 Results

Following the methodology described in the previous section, 17 online articles, blogs, security advisories and publications were collected and analyzed. Following is a list containing a short summary of each article considered for this component.

Interpreting Kafka Incidents — KAFKA-15096/15498/14800 [1] In this article, the authors introduce an Apache Kafka vulnerability, identified by CVE 2023-34455 [14], in which a malicious actor with access to the packets sent through the network to an Apache Kafka client (broker/server/client proper) can cause an *OutOfMemoryException* in the Java VM running the software. This issue was caused by a flawed version of the `snappy-java` library, `v1.1.8.4`. More specifically, a malicious threat actor could modify the packet bytes of a compressed message to indicate to the Kafka client that the message's size was any arbitrary value. The decompression algorithm implemented by the vulnerable version of the `snappy-java` library failed to perform a bounds check, which could cause unnecessarily large data arrays to be allocated, leading to the aforementioned error. Solving this problem involves updating the Apache Kafka installation used to a version which does not depend on the affected library (i.e., versions released after version `3.5.0`).

Critical Apache Avro SDK Flaw Allows Remote Code Execution in Java Applications [49]

The author of this article describes a critical flaw discovered in Apache Avro[29], a serialization format and an ecosystem of libraries built around that format. More precisely, the flaw was discovered in Apache Avro's Java SDK, having been identified in CVE 2024-47561 [17]. The main functionality of Avro is centered around *Schemas* which dictate the shape any data should take. Due to an unchecked attribute in versions of the software until version 1.11.4, arbitrary code could be loaded into a JVM's memory and executed, leading to a case of Remote Code Execution (RCE). This only happened when an application allowed arbitrary schemas to be supplied by users. A solution for this problem revolves around updating the Kafka installation to use a version of the Java SDK that is not vulnerable.

CONFSA-2023-03: Confluent Platform Vulnerability in Schema Registry: Schema links can be misused to bypass Role Based Access Control (RBAC) based read restrictions on Schema Registry [2]

In this advisory, a vulnerability in the Confluent Platform product was identified, particularly with the *Schema Linking for Confluent Platform* feature. This affects several versions of Confluent Platform (7.0.0–7.0.8, 7.1.0–7.1.6, 7.2.0–7.2.4, 7.3.0–7.3.2). The aforementioned feature allows one Confluent Schema Registry to export *subjects* and *contexts* (groupings of subjects) to another Schema Registry. Prior to the release of Confluent's *RBAC for Schema Linking* feature, a user without privileges to access or modify schema subjects and contexts could exploit the Schema Linking feature by exporting these same entities to a cluster where they have the required privileges. This vulnerability only affected entities secured under read permissions: write, delete and update operations were not affected. According to the article, remediation starts by upgrading to a patched version of Confluent Platform (7.0.9, 7.1.7, 7.2.5, 7.3.3). Afterwards, a thorough configuration audit should be performed to make sure no existing configurations could still allow for malicious actions.

Reducing the Insecure Deserialization Risk - Cyber Defense Magazine [30]

The author of the articles aims to shed a light on a prevalent category of cyber-security problems: deserialization issues. Errors based on deserialization occur when an application receives untrusted data and parses/deserializes it without properly verifying the integrity of the data. This could lead to malicious code being executed on the environment of the application doing the deserialization. According to article, even though this issue mainly plagues Java applications, all languages that include deserialization capabilities in their standard libraries can be affected by this problem. Types of Java deserialization attacks include: *Blind attacks*, where an attacker aims "to extract data from the target system in environments where the system is behind a network firewall that blocks outgoing connections"; *Asynchronous (or stored) deserialization attacks* where malicious code is stored on the target system and is executed eventually when the system reads back that data; *Deferred-execution deserialization*, in which malicious data does not directly trigger code execution, which happens during cleanup tasks (such as the `finalize` method in Java objects). The best remediation for this type of vulnerability is to thoroughly patch the vulnerable code. However, if this is

not possible, maybe because the vulnerability lies in some dependency or third-party component crucial to the correct operation of a system, strategies such as *Whitelisting*, *Blacklisting* or the use of *Security Managers* could provide an extra layer of security to applications and systems, albeit not totally, as is explained by the article's author. It is theorized that the *de facto* way to prevent deserialization issues from occurring is to prevent deserialization altogether. It is worth mentioning that, while this article does not explicitly relate to Event-Driven Systems, it refers to Java applications, an example being Apache Kafka, and as such this article is deemed relevant for this research.

Why You Need to Look Beyond Kafka for Operational Use Cases, Part 3: Securing the Eventing Platform | Solace [6]

In this article, the author provides a side-by-side comparison between Apache Kafka and their own software product, Solace, highlighting the key advantages of Solace when compared to Apache Kafka. They describe how Solace offers advanced security features natively while achieving the same with an Apache Kafka installation is far from trivial. Other apparent disadvantage of a Kafka installation compared to Solace's solution is the insecure nature of adding security features to a Kafka cluster, which they exemplify by arguing that adding extra security services, like business grade authorization and authentication services, relies on insecure, and thus, untrusted network connections. While the article itself serves more as a marketing strategy, it does highlight some security aspects, albeit outdated, of Apache Kafka, making this article useful in figuring out possible attack vectors.

8 Tangible Ways — Including Least Privilege — to Secure Event-Driven Software Architecture [23]

In this post, the author presents some general tips about securing a system built around an Event-Driven architecture. At a first instance, a brief contextualizing explanation about what Event-Driven architectures are is provided, after which the main problems regarding securing systems built on top of an Event-Driven architecture are presented. One prime example given of applications affected by these issues is Apache Kafka. Security measures are then described, including: *implementing strict input validation and parameterization*, which ties in to the previously mentioned issues regarding deserialization (which stem from invalidated inputs being processed by the system); *implementing secure communication channels* so that data-in-transit is secured against any malicious actors listening on the network, possibly preventing man-in-the-middle attacks; *enabling proper logging and monitoring* to track event workflows and enabling tracing back faulty/suspicious operations; *using secure authentication and authorization mechanisms* to ensure that only the right users have access to the right resources, such as event brokers or other system components; *enabling authorization of read/write operations*, which goes hand-in-hand with the previous point, to prevent resource misuse/overuse by malicious actors; *leveraging security features of the event streaming platform* which provide an established base for improving the security posture of an Event-Driven System without much effort needed from the implementers (since part of the security system is already implemented by the platform itself); *knowing a system's surface area*, allowing for more informed security decisions regarding the system itself. Furthermore,

some tools and techniques useful for achieving those measures are presented, such as API gateways for security policy enforcement, IAMs for authentication and authorization management or IDS or IPS systems which can detect or prevent, respectively, intrusions in a network that could negatively affect the system. Despite providing only very general tips, this are nonetheless useful in the context of securing Event-Driven Systems.

Lessons Learned From The Trenches Developing Event-Driven Systems [46] In this article, the author makes the point that correctly implementing an Event-Driven Systems is harder than what it is made out to be. They argue that the common solution to Event-Driven Systems is effectively limited to a single *webhook*, which in turn makes it entirely wrong. They present 2 approaches to building Event-Driven Systems: *Stream/Webhook and Data Dump* involves a system, System 1, propagating events through the entire architecture, possibly mediated through an event bus, when some modifying operation happens, which would then be picked up later by a second System, System 2; *Event Sourcing*, where state altering operations are not persisted to the underlying data store directly, instead being encoded inside *events* which are added to an append-only table or log, which by definition is ordered by event creation and thus allows replaying those same events to "replay" the state-modifying operations that occurred. Event Sourcing is presented as the better alternative to building Event-Driven Systems due to its high scalability potential. Despite all these, it is noted that a *periodic reconciliation* process is needed to ensure consistency of data across the entire software stack of the system. Several other necessary features that increase the complexity of these kinds of systems, such as *tombstone records*, *optimistic locking for updates* and *retry storms* are presented.

Common Kafka Security Misconfigurations and How to Avoid Them [65] In this article the author enumerates several security misconfigurations in systems using Apache Kafka and, for each one of these, a prevention method is presented. In order of appearance, the mentioned security issues are: *misconfigured authentication*, where a Kafka installation was properly setup and the system is operating as expected but, due to a bad authentication configuration, access to the Kafka cluster is public, with solutions including setting up common authentication methods like *Simple Authentication and Security Layer* (SASL) and performing regular audits to the system's security configurations; *weak or missing ACLs*, where improper authorization control is configured and so there is no security mechanism enforcing restrictions on data access, preventable with regularly audited ACLs and *Role-Based Access Control* (RBAC) concepts encoded in those ACLs; *lack of encryption for data in transit*, which paves the way for malicious actors to gain access to data in transit on the network which should be secured using mechanisms like *SSL/TLS*, with certificate expiration being frequently monitored; *insecure data at rest*, consisting of a lack of encryption and security for data stored at rest by the broker, which could be exposed through physical access to the physical data storage devices on which the broker runs, secured through setting up string encryption for resting data with secure or rotating security keys; *misconfigured replication settings*, which can lead to operational issues and data loss if large system traffic is not accounted for and

fixed by properly configuring the replication properties of the Kafka, such as a topic's replication factor or the number of minimum In-Sync replicas; *poor monitoring and auditing practices*, which involve a lack of awareness of events, faults and other relevant issues that might occur within a system and which might come to light with the implementation of audit logging for Kafka, proper resource and operational monitoring and real-time alerts for various critical system resources. A final remark about the dynamic and constantly updating nature of system configurations is made by the author.

Kafka Unauthorized Access | Shoreline Runbooks [64] This web article consists of a summary of a common problem in Kafka installations, *misconfigured ACLs*, which might lead to unauthorized accesses to data and other, possibly modifying operations on the cluster's resources. Diagnostic and remediation steps are then promptly introduced.

Rabbit MQ Queue Policy Mismatch [47] This article introduces *Queue Policy mismatches*, describing them as an error that occurs when there is an attempt to apply a policy to an existing queue but the policy's parameters conflict with the queue's current configuration. This conflict can lead to message flow disruption and ultimately semantically undefined behavior. Causes for this mismatch might be attempting to mirror a queue that is not configured for mirroring or setting a Time-To-Live (TTL) that is incompatible with the existing TTL settings. Resolving these types of issues revolves around carefully analyzing the new policy and adjust any invalid values.

Misconfigured Kafdrop Puts Companies' Apache Kafka Completely Exposed [48] In this article, the authors present a problem of exposed Kafka clusters via a second application, Kafdrop, an Open-Source UI app for interacting and managing Kafka clusters. Despite there being no vulnerability in any of these components, their interaction exposes Kafka clusters publicly, potentially leading to catastrophic breaches and data leakages. Since Kafdrop provided public access to itself, requiring no user authentication, any exposed Kafdrop deployment would indirectly expose the underlying Kafka cluster. Remediation can start with exposing Kafdrop through a reverse-proxy server, like Nginx, and configuring this proxy with adequate security modules and considerations.

Pentesting Apache Kafka 101 : Top 5 Security Misconfigurations [11] In this article, the authors explore several security misconfigurations in Apache Kafka installations using Zookeeper as a configuration store. An initial contextualization of what Event-Driven systems are and how Kafka might enable those systems is given at the beginning, being followed by an enumeration of the several identified security issues: *insecure authentication and authorization* by using, for example, *plaintext* authentication, which makes the cluster transmit credentials without any encryption, which should be used instead (with common schemes being TLS, SASL, etc.); *lack of encryption*, meaning that any data transmitted between cluster components is transmitted in plaintext and as such can be exposed to malicious actors. This can be avoided using TLS as an encryption method for inter-cluster communication and communication between the cluster and

clients; *lax access control policies*, related to misconfigured ACLs, which enable unauthorized access to all of a cluster's resources, potentially leading to destructive operations, which can be prevented by enabling authentication and authorization modules and configuring Access Control Lists by following the *principle of least-privilege*; *improper tuning of consumer configurations*, particularly those related to timeouts and retries, which can lead to a consumer being overloaded and thus the system experiences a Denial-of-Service. Mitigation revolves around properly analyzing and configuring values such as the max polling interval for consumer clients; *inadequate log retention and cleanup* which, since Kafka is a distributed-log by definition, would mean inadequate storage of message logs, potentially exhausting a host's resources, such as available disk space, and potentially leading to a Denial-Of-Service if preventive measures like proper log retention configuration are not taken; *exposed ports and unrestricted network access*, due to Kafka's default behavior of listening on all available interfaces, which can be remedied by manually specifying the interfaces Kafka should use. Further "hardening considerations" are provided, such as using custom scripts and load testing tools to regularly stress the system with hopes to occasionally find some hidden corner-case of the system, or performing regular security audits to the entire Kafka installation to make sure an organization's deployment conforms to a vendor's security best practices and guidelines.

Security Bulletin: Security vulnerability in Apache Kafka clients affects IBM Business Automation Workflow Case Event Emitters - CVE-2024-31141 [34] In this security advisory, a security flaw affecting IBM's Business Automation Workflow Case Event Emitters was discovered to be caused by a flaw in the packaged distribution of Apache Kafka. Identified by CVE-2024-31141 [15], this flaw lets attackers exploit a vulnerable Kafka installation via a feature called *ConfigProviders*. These provide an interface for clients to specify system configs by their preferred method, whether one of the pre-defined providers or a custom provider loaded by the clients at runtime. Unrestricted access to these providers can allow unauthorized access to filesystem/disk content or environment variable data, which may contain sensitive information. In particular, users of Kafka Connect can suffer from escalation of REST access to filesystem/environment access. This issue affects Apache Kafka versions 2.3.0–3.5.2, 3.6.2, 3.7.0. Remediation can be achieved by upgrading the used Kafka dependency to versions $\geq 3.8.0$. Further mitigation involves setting a special JVM system property to restrict automatic config provisioning, which will thus need to be explicitly controlled by deployers.

Critical Alert: CVE-2025-27817 in Apache Kafka Client [58] In this article, the author discussed a previously disclosed CVE, CVE-2025-27817 [18], and the security implications of a potential fix release by the Kafka development team. The original CVE relates to a security flaw in Apache Kafka where sensitive configuration parameters, namely `sasl.oauthbearer.token.endpoint.url` and `sasl.oauthbearer.jwks.endpoint.url`, could be used to escalate privileges if controlled by an attacker. By setting these configuration values, a malicious actor could access and read

arbitrary local files or environment variables. Furthermore, they could set these configuration values to an attacker-controlled server, making the Kafka system send requests to an unauthorized location. This could give attackers full control over the system, leading to potential Server-Side Request Forgery attacks. Version 3.9.1 aimed to fix this issue by introducing a new JVM property, `org.apache.kafka.sasl.oauthbearer.allowed.urls`, which controls the allowed values for the previous configuration values, but this value is kept blank in this release for backwards compatibility reasons. The author of the article argues that this solution still remains insecure due to a lack of "security by default" and the need of operators to manually set this property to the correct values, possibly giving a false sense of security to organizations that just bump the affected package's version number. Instead, users should use versions $\geq 4.0.0$ which completely fixes this vulnerability by forcing users to explicitly set the allowed values for those config values, thus removing the insecure default configuration of the previous version. Despite this advantage, it is recognized that a migration process from versions $3.x.x$ to $4.x.x$ need to be carefully planned, due to potential breaking changes introduced in the new version.

CVE-2024-32030 [16] details how Kafka UI, an Open-Source Web UI for Apache Kafka, can be used by malicious threat actors to access Kafka brokers. Kafka UI is able to monitor the performance of Kafka brokers through their JMX interface, which is based on Java's RMI API, making it "inherently susceptible to deserialization attacks". This feature could be exploited by a potential attacker by connecting Kafka UI to an attacker-controlled broker. By crafting a malicious RMI object that is returned on any RMI call, an attacker can inject custom malicious code in Kafka UI's runtime and gain privileged access to the system. Possible consequences include *post-auth* remote code execution. This is enabled by the fact that Kafka UI contains in its classpath the required gadgets to perform the attack. The issue becomes aggravated by the fact that Kafka UI does not enforce authentication by default. Users are advised to upgrade to versions $\geq 0.7.2$ since there is no known workaround available.

Apache Kafka Flaw Let Attackers Gain Access To Sensitive Data [25] In this article, the author exposes a vulnerability affecting Apache Kafka installations that are migrating from Zookeeper to KRaft as a configuration strategy. More precisely, this vulnerability affects ACL enforcement in clusters where this migration is underway. If 2 preconditions are met, 1. administrators removing an ACL for a given resource, 2. the associated resource must have 2 or more ACLs after removal of the first one, then this bug can trigger, causing only the first ACL found by Kafka for that given resource to be enforced. The vulnerability is cleared once the migration is complete and the system's knowledge of the resource's ACLs is stabilized again, for example by adding a new ACL to that resource. The severity of this vulnerability will depend on the nature of the ACLs being ignored. Generally, if the ignored ACLs referred to `ALLOW` actions, the impact is lessened since all operations could be denied, and thus malicious operations would not affect the system. If, however, the ACL referred to a `DENY` action, this vulnerability could "escalate to affected confidentiality and integrity" since a malicious actor could now perform operations that were hitherto

blocked. Affected versions include 3.5.0–3.5.2 and 3.6.0–3.6.1. Fixing the vulnerability is a matter of upgrading the vulnerable Kafka installation to a later version.

CVE-2025-27818: Apache Kafka: Possible RCE attack via SASL JAAS LdapLoginModule configuration [9] This mailing list entry details a possible security vulnerability identified in Apache Kafka and Kafka Connect software components. Given access to configuration altering operations (either via config files or AlterConfig requests) or connector creation via the Kafka Connect REST API, an attacker can set the value of the `sasl.jaas.config` configuration property to `com.sun.security.auth.module.LdapLoginModule`. This change can allow a Kafka client or Kafka Connect connector to connect to an attacker-controlled LDAP server. By providing a malicious LDAP payload that is deserialized by Kafka components, the attacker can execute deserialization gadgets available on the Kafka server and potentially gain access to the vulnerable infrastructure, which can lead to Remote Code Execution attacks. This vulnerability exists in versions 2.3.0–3.9.0, and updating to versions 3.9.1 or 4.0.0, which introduce a new system property for blacklisting potential login modules, should solve this issue.

5.1.4 Categorization

Compiling the results obtained and described in Section 5.1.3 and in accordance to the methodology described in Section 5.1.2, 4 main categories of vulnerabilities in Event-Driven Systems were identified: *Broken Access*, *Serialization/Deserialization* (referred to as *Ser/De*), *Policy Misconfiguration* and *Security Misconfiguration*. For each category, a description of the kinds of errors that fall into them is given, as well as examples of attacks and prevention mechanisms, much like what is done in OWASP's Top 10 ranking. Links to the relevant articles are included.

It is worth noting that *Broken Access* can be subdivided into further sub-categories, such as *Improper Privilege Management* and *Policy Enforcement*, each of which have their own articles supporting them. Despite this, since they ultimately fall into the wider category of Broken Access, they are all accounted in there. Further explanations are given on a per-category basis. Furthermore, many of the categories present also exist in the "non-Event-Driven" rankings developed by OWASP, such as the main Top 10 ranking. However, in this document they are described and their findings contextualized to Event-Driven Systems.

Out of the 17 publications read, one of them did not present sufficient data to infer a category of vulnerabilities. Nonetheless, it is still included for statistical purposes, such as calculating the percentage of incidence of a certain category.

5.1.4.1 Broken Access

Description By far the most detailed category, with 11 reported instances throughout the reviewed articles (64.7%), this category is related to issues where malicious actors gain some level of access to sensitive or protected resources. Access control is the mechanism by which resource access is mediated through a system which verifies both the actor attempting to access the resource

and which operation they want to apply on it. Having a broken access control mechanism opens the door to several attacks which can harm business operations and customer trust.

From the corpus of analyzed articles, some sub-categories could be identified for Broken Access:

- **Improper Privilege Management** is related to mismanagement of user and system privileges, granting access to restricted or sensitive information those entities wouldn't otherwise have, this integrating the Broken Access category.
- **Policy Enforcement** includes errors in enforcing behavior via policies, leading to authorization mechanisms operating in a faulty manner and thus potentially allowing access to untrusted entities.

Examples

- **Scenario #1** An application parses some untrusted data without performing proper verification. Depending on the supplied payload, code injection might happen, which in turn might enable remote access to an attacker, which is now free to move horizontally throughout the vulnerable system.
- **Scenario #2** Due to some misconfiguration in the authorization rules/mechanism of an organization's system, certain resources are publicly exposed, where every user (good-willed or not) may access and see them. If these contain sensitive information, it is counted as a serious data leak.

Mitigation More often than not, the recommended mitigation action is to upgrade affected system components from the vulnerable version into a version where the vulnerability has been patched. This recommendation is more important in situations where the vulnerable component is not the system itself but one of its dependencies, in which case the vulnerable status of the system exists as long as the vulnerable versions of the software are used as a dependency.

More detailed mitigation steps include frequent security audits to authorization policies and authorization processes (authorization configurations, authorization code, etc.) and use of secure and trusted third-party products/services for access management, such as IAMs.

Referenced Articles [2, 6, 11, 23, 25, 34, 47–49, 58, 64]

5.1.4.2 Deserialization Vulnerabilities

Description Serialization is the process of converting structured application data (such as an object or some other form of data representation) into a format that can more easily be stored or transmitted. Conversely, deserialization, discussed in 29.4% of the reviewed articles, is the process by which previously serialized data is consumed to construct structured application data. Depending on the system, the runtime environment or deserialization library used, some form

of code execution might happen upon deserialization or as a result of object deserialization, like instantiating some data that could not be transmitted/serialized and should be fetched before any more processing can be applied to that data. If a malicious actor can control the data that will be deserialized, they can inject malicious subroutines or trigger gadget chains in the target system. This opens the door to a multitude of attacks that can be done on the target system, such as Denial-of-Service attacks and Remote Code Execution (RCE).

Examples

- **Scenario #1** An application allows its users to input some data which is then read by the system in some way. A malicious user, with enough resources, could gain an understanding of the technologies used in the system, thus gaining knowledge of potential vulnerabilities affecting the system. If proper input validation is not set up, a malicious payload could, upon deserialization, trigger some gadget chain available in the application's code that leads to remote execution of code.
- **Scenario #2** An application reads a stream of data that respects a given protocol. This protocol specifies that some portion of the data stream represents the length of some content. If proper validation is not used, a malicious actor could modify this value to a number which far surpasses the host's resources, potentially leading to the allocation of extremely large buffers and causing resource exhaustion, eventually leading to a Denial-of-Service (DoS).

Mitigation As with the previous category, the most common strategy to securing a software system against deserialization issues is to make sure all components that participate in the deserialization process are up to date and aren't themselves vulnerable. This is specially important if the deserialization vulnerability happens in a software dependency, meaning the system's deserialization vulnerability is entirely dependent on whether a fix has been released for the affected dependency or not.

Other, more manual steps to ensure protection against deserialization vulnerabilities include regular reviews of code that is responsible for parsing and deserializing external, possibly untrusted, input and making sure that all parameters that might cause issues to the system if they were modified by an attacker are properly checked. A more extreme solution that has been presented in one of the reviewed articles but understandably cannot be applied to a majority of systems is to disable deserialization altogether and, this way, preventing this category of vulnerabilities from surfacing entirely.

Referenced Articles [1, 16, 30, 49]

5.1.4.3 Policy Misconfiguration

Description Policy Misconfiguration, referenced in 17.6% of reviewed articles, involves setting up authorization mechanisms, which act based on configuration objects which can be thought of as

implementing a policy, in such a way that they do not respect their intended use. By preventing legitimate operations (false negatives) or allowing unauthorized access (false positive) to resources, vulnerabilities in this category can lead to serious problems for an organization, including data breaches and threat actors maliciously affecting the companies operations through methods which should prevent those events by definition. Examples of enforcement of policies can be found in Kafka through their Access Control Lists (ACLs) feature or in RabbitMQ with their Queue Policies.

It should be noted that policies and policy enforcement are often used as a gate, separating the user from other parts of the system. As such, an attack that exploits Policy Misconfiguration usually targets other vulnerabilities, such as Broken Access.

Examples

- **Scenario #1** An operator sets up ACLs in a company-wide Apache Kafka installation, but mistakenly allows read accesses for a given topic for any user (represented as a wildcard access), meaning that anyone with access to the cluster, even employees who should not be able to read its contents, can exfiltrate data for malicious purposes.
- **Scenario #2** in a RabbitMQ installation, some queues were configured with a given policy. Later, while reviewing changes in their system, these queues need newer permissions and access control. An operator attempts to apply an incompatible policy, leading to downtime. If the policy is compatible but incorrectly implemented, it could expose the queue to the public, possibly allowing uncontrolled unauthorized access to its data.

Mitigation To remediate Policy Misconfigurations, regular audits to the policy configurations, be it Kafka ACL definitions or RabbitMQ Queue Policies, should be performed against a large and randomized set of values to stress whether some uncommon corner case that violates the policy's contract exists. Companies are advised to have personnel dedicated to policy management and maintenance, as this also helps keep organizations secured against updating security standards and business requirements.

Referenced Articles [11, 64, 65]

5.1.4.4 Security Misconfiguration

Description With only one article reporting security misconfigurations (5.8%), this category encompasses all vulnerabilities that compromise a system's security directly but were not caused by an insecure system design (as could be seen with *Broken Access*) nor specialized mechanisms which can be purposed for security reasons (like authorization issues arising from *Policy Misconfigurations*). Even though there's only one article related to this category, a lot of vulnerabilities should match this category since it is so broad.

Examples

- **Scenario #1** a Microservices system uses HTTP for the internal communication between its components. While this is faster, since it skips the TLS handshake, certificate verification and key exchange steps, it opens up the possibility that an external, possibly malicious actor, who somehow got access to the system's internal network, can sniff out packets as they are being sent. Since they are transmitted in plaintext, the attacker can read all the exchanges being made between services, possibly containing sensitive information.

Mitigation Properly configuring security features of the various system components being used, such as in-transit data encryption, is essential for assuring that a system maintains a security posture that is as strong as possible against malicious external threats.

Referenced Articles [64]

5.1.5 Threats to Validity

As discussed in Section 5.1.2, many of the analyzed articles from which the categories described in the previous section were defined contain overlapping dimensions. It is worth mentioning that the interpretation of all articles was done by the author singlehandedly. Not only does this mean that the data interpreted might not be what the original author of an article meant to expose, it also indicates that any bias inherent to the analysis process got introduced in every analysis made. Another problem with the analysis and methodology taken is the number of analyzed articles: it could be argued that it is too low a number from which to draw valid conclusions in the context of a vulnerability taxonomy. While this is recognized as a shortcoming, efforts were made to explore as much of the gray literature as possible without repeating articles, which was not always the case. More specifically, while collecting articles for analysis, for several different queries, one of two things happened: 1) the same articles could appear multiple times on the first page of results; 2) different articles related to the same CVE, vulnerability or security advisory would appear. Finally, the categories themselves overlap in some ways, with articles sometimes belonging to two or more categories. This can mean that better care should be taken in order to correctly distinguish all categories available.

To the best of our knowledge, no other threats to the validity of the methodology described in this section could be found.

5.1.6 Summary

In this section we introduced a brand new taxonomy of Vulnerabilities and Security Risks in Event-Driven Systems, taking inspiration from established documents like OWASP Top Ten and OWASP Cloud-Native Application Security Top 10 [53, 55]. As described in Section 5.1.2, using search queries made up of relevant words in the context of Event-Driven Systems, searches were performed on internet search engines, with the first page of results being collected for further

analysis. Duplicate publications were excluded from this collection process. Article analysis was performed according to 4 main dimensions (the *how*, the *when*, the *where* and the *why*) and is presented in Section 5.1.3. From this, a taxonomical categorization of vulnerabilities in Event-Driven Systems is introduced in Section 5.1.4, containing 4 main categories of vulnerabilities: *Broken Access*, *Deserialization Issues*, *Policy Misconfiguration* and *Security Misconfiguration*. Finally, threats to the validity of the article analysis process were detailed in Section 5.1.5.

5.2 EDA Vulnerability Threat Model

5.2.1 Context

Given the taxonomy of Event-Driven vulnerabilities described in the previous Section, we now set out to construct a suitable Threat Model based on that taxonomy. This involves collecting vulnerabilities described in the articles analyzed in the previous section and attempting to replicate the associated exploits whenever possible. When replication is not possible or is otherwise unfeasible, accounts from the analyzed articles will be used as best as possible to provide a detailed entry in the Threat Model. Adding to the results in this section are other experiments, based on known vulnerabilities and exploits but which aren't present in the corpus of analyzed articles.

In an attempt to provide a list with some structure based on already established work, inspiration will be taken from OWASP's Threat Modeling Project[54]. By following the conventions and framework already in use, we can develop a Threat Model that can be understood and processed immediately by individuals who already work with that format. To this end, the Threat Model presented in this section will be structured according to the following items, adapted from the official OWASP Threat Modeling Project website¹:

- Description of the systems we are "worried about" (Event-Driven Systems, in this case);
- List of assumptions that should be checked as the threat landscape surrounding Event-Driven Systems evolves;
- List of potential threats to Event-Driven Systems, derived from the aforementioned taxonomy and practical experiments conducted;
- A list of actions to be taken to mitigate/remediate each threat's consequences;
- A way of validating the model and threats, and verification of success of actions taken;

Before the proper Threat Model is introduced, we first establish the base test environment used throughout the experimentation process. Experiment-specific modifications to this environment are introduced on a per-experiment basis.

¹<https://owasp.org/www-project-threat-model>

5.2.2 Test Environment

In order to properly assess each vulnerability, its exploits and mitigation steps, a suitable environment must be configured. For this, we used Confluent’s Docker images as a base test bed. The base experimental configuration is based on Confluent’s official guide for quickly setting up a Confluent Platform installation running locally on Docker containers²³, with some of the less important services removed for clarity. For brevity, only a diagram description, seen in Figure 5.1, is included in this chapter, with the actual configuration present in Appendix B.1.

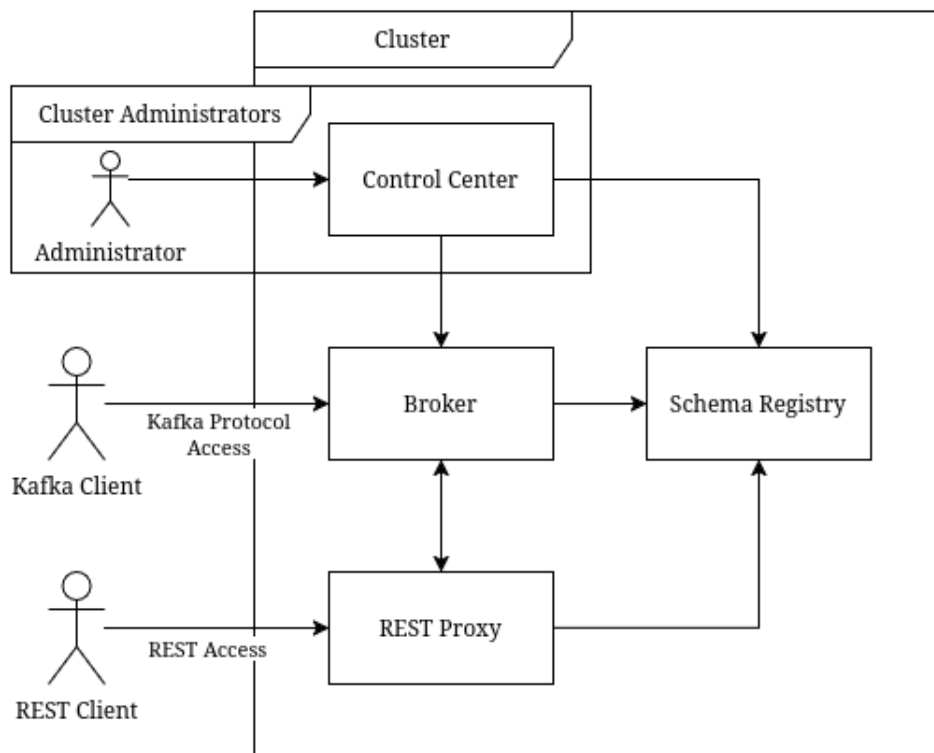


Figure 5.1: Diagram of base setup for experiments

5.2.2.1 Experiment 1: Deserialization of Avro-encoded data with a static schema

Apache Avro is a data format and a library aimed at simplifying data transmission and processing. It is a first class citizen in the Kafka ecosystem, being the main Schema library used to ensure data validity in topic messages.

For this vulnerability, we aim to assess how to explore and exploit a Java application using a vulnerable version of the Apache Avro Java SDK like the one identified in [49]. This can serve as a base for further expanding this exploit to environments running Kafka (which is a Java application). No modifications to the Docker environment were made. However, a file,

²<https://docs.confluent.io/platform/current/get-started/platform-quickstart.html#ce-docker-quickstart>

³<https://github.com/confluentinc/cp-all-in-one/blob/8.0.0-post/cp-all-in-one/docker-compose.yml>

`bad_schema.avdl`, written in a custom language meant to more closely resemble common programming languages, was added. Listing 5.1 presents the contents of this file in full.

```
1 namespace org.example;
2 schema Exploit;
3
4 record Exploit {
5   @java-class("org.example.ExploitClass") string entrypoint;
6 }
```

Listing 5.1: `bad_schema.idl`

Through a custom script, this file is transpiled into an Avro schema file and, from there, into an actual Java class that can be used in the codebase. The modifier `java-class` means that a custom Java class (`org.example.ExploitClass`) is used as the actual datatype of the `entrypoint` attribute. Simply creating this class with some form of malicious code (in this example, a standard output log showing that custom code was ran) and placing it into the application's classpath is enough to trigger the vulnerability.

While it is possible to trigger this vulnerability, with the possible consequences of it ranging from harmless behavior like the one displayed in the experiment to possible reverse shells or Remote Code Execution, it should also be taken into consideration the multiple factors needed for this exploit to be successful: the target system must be using, in their IDL files or Schema files, the `java-class` attribute; the target system must be statically compiling their Avro schemas to Java code; the attacker must know the fully-qualified name of the Java class used as the data type for that attribute; the attacker must have access to the application's classpath (probably via the application's supply-chain) in order to inject the necessary malicious code. It quickly becomes clear that this exploit, while extremely dangerous, is also extremely hard to accomplish.

Further experimentation on the same Avro Java SDK RCE was attempted, this time providing a dynamic schema along with data being deserialized. The idea is that, by providing the schema along with the data, there would be no need to go through the code generation process of the previous experiment, simplifying the overall exploit. However, no successful attempt has been made and as such this experiment is not considered.

Both of these experiments can be extended to systems deploying Apache Kafka when these use Apache Avro as a schema library. If external schemas can be provided, an attacker only needs to gain control of Kafka's classpath (via the system's supply-chain mechanisms, for example) to enact the rest of the exploit.

5.2.2.2 Experiment 2: Allocation of a memory buffer from Snappy-compressed data

Apache Kafka offers multiple compression strategies for big messages. One of these strategies uses the `snappy-java` library, a Java port of the `snappy` library developed by Google.

For this vulnerability, we aim to assess how to explore and exploit a Kafka deployment using a vulnerable version of the `snappy-java` library like the one identified in [1]. Using `snappy` compression, the wire protocol includes in its data the length of the decompressed data. A malicious attacker with access to the request data can modify this parameter and force the vulnerable version of the library to allocate a buffer with arbitrary length, such as a number bigger than the host's maximum memory, thus causing an *OutOfMemoryException* and disruptions to the service and eventually leading to a DoS.

The changes made to the base environment are presented in Listing 5.2.

```
1 services:  
2   broker:  
3     ...  
4   environment:  
5     ...  
6     KAFKA_COMPRESSION_TYPE: gzip  
7     ...
```

Listing 5.2: Base environment changes for `snappy-java` experiment

Together with this changes, a script that manually sends message data back to Apache Kafka was also developed to attempt and intentionally trigger the vulnerability. The script manually assembles the binary data that is sent to the broker according to the Kafka protocol and reverse-engineered traces of network packet exchanges viewed in Wireshark⁴, a network packet sniffing tool. By setting the compression type to indicate that `snappy` compression was used and carefully crafting the compressed payload, an *OutOfMemoryException* could be triggered which would crash the broker instance

In reality, the value of the environment value used in the environment configuration could be any of the allowed values for that attribute except `snappy` since this mismatch between broker compression type and topic compression type triggers full topic decompression and re-compression, potentially also triggering the vulnerability.

Despite our best efforts, no successful experiment could be achieved in none of the tired configurations. Despite errors having been reported by the Kafka Broker, these were most likely protocol-related errors, indicating that some aspect of the exploit script is flawed. Furthermore, there is no available exploit specifically targeting Apache Kafka. [1] only introduces an example of exploiting this vulnerability directly on the `snappy-java` library.

Nonetheless, the author of the article makes the claim that this vulnerability can be triggered in Apache Kafka via an exploit of the `snappy-java` library. This claim will be taken into consideration but with the knowledge that, since these claims could not be confirmed, they could be wrong or outright false.

⁴<https://www.wireshark.org/>

5.2.2.3 Experiment 3: Code execution via policy authorization

Apache Kafka offers a built-in authorization framework that allows custom policies, in the form of Access Control Lists (ACLs), to be applied and either deny or allow a given operation for a given resource. The developers of this authorization framework made it so that it was highly extensible, allowing for external, custom code to be used as the authorizer. Examples of when this modularity is useful is when a company uses an external authorization engine for which no suitable adapter is available and so they have to build their own adapter in-house.

For this vulnerability, we aim to assess how to explore and exploit a Kafka deployment using injected malicious code in a policy authorizer. Using a custom-made policy authorizer, containing some "malicious code" (in this case just statement logging to standard output and a dedicated file for easier verification), that is injected in the broker's classpath, upon any authorization request the malicious code present in the authorizer is executed. The impact of this exploit can be negligible, as is the case in this experiment, or can be catastrophic, with experienced attackers being able to execute code remotely and potentially gaining access to the Kafka broker's host.

The changes made to the base environment are presented in Listing 5.3.

```
1 services:  
2   broker:  
3     ...  
4   environment:  
5     ...  
6     KAFKA_CREATE_TOPIC_POLICY_CLASS_NAME: org.example.BadPolicy  
7     KAFKA_ALTER_CONFIG_POLICY_CLASS_NAME: org.example.BadPolicy  
8     CLASSPATH: "/plugin/*"  
9   volumes:  
10    - "./plugin/:/plugin/"  
11    - "./logs/:/logs/"
```

Listing 5.3: Base environment changes for custom authorizer experiment

As is visible in the configuration changes, a directory `plugin` was created to store the compiled code in JAR (Java Archive) format. Furthermore, the `logs` directory is used to store a log file generated by the bad authorizer code for easier verification of the exploit's success.

Looking at an excerpt of this log file, it is clear that injected malicious code is being executed for every authorization request. This excerpt can be found in Appendix B.1. One advantage that this experiment enjoys over the rest is that exploit triggering happens automatically, without manual intervention. This is due to Confluent Platform (and even the base Kafka distribution) performing maintenance operations on its cluster data and topics, leading to validation requests being made automatically and consequently triggering the malicious code.

However, as can be seen by the environment changes made for this experiment, the classpath of the Kafka broker was altered. This indicates that access to the supply chain is needed to execute this exploit. Furthermore, unless an attacker knows beforehand that a system is using custom policy authorizers and knows the their fully qualified names, it should prove extremely difficult to enact this exploit.

5.2.2.4 Experiment 4: Unauthorized operations via ACL bypass

Kafka uses Access Control Lists (ACLs) as *authorization guards*, making sure that a *principal* (i.e., an entity authenticable against a Kafka Broker) is only allowed certain operations on certain resources when allowed by system administrators or operators responsible for maintaining an organization's access control directives. Principals include users and users groups for the default authorizer, but custom authorizer implementations may provide different principals. A resource includes topics, cluster metadata, among others. Operations include read, delete, describe, create, etc. An ACL can thus be thought of as a 4-tuple in the form (*principal*, *operation*, *resource*, DENY|ALLOW). Some of these elements, like *principal* and *resource*, may include *wildcards*, meaning the ACLs they are a part of apply not to a single entity but to a whole category of entities. Some ACL operations imply others, as is the case with the DESCRIBE operation being allowed if any of the READ, WRITE or DELETE operations are allowed for a given principal. In contexts where both a DENY ACL and a ALLOW ACL exist, DENY ACLs take precedence.

For this vulnerability, we aim to assess how to explore and exploit a Kafka deployment by executing actions which semantically are disallowed for our principal but are not covered by the existing ACL mechanism. ACLs in Kafka are configured using the `/bin/kafka-acls` auxiliary tool. Due to the amount of configuration options needed to properly run this tool, an auxiliary script was created to ease command execution:

```
1 #!/bin/env bash
2
3 EXECUTABLE=/bin/kafka-acls
4 BROKER_HOST=localhost
5 BROKER_LISTENER_PORT=9092
6 BOOTSTRAP_SERVER="$BROKER_HOST:$BROKER_LISTENER_PORT"
7 COMMAND_CONFIG="/etc/kafka/kafka.properties"
8
9 "$EXECUTABLE" --bootstrap-server "$BOOTSTRAP_SERVER" --command-config "
    $COMMAND_CONFIG" $@
```

Listing 5.4: Auxiliary ACL tool

This vulnerability has the potential to pose serious to an organization or system should an attacker gain access to administrative tools, because it can directly mean the disruption of normal service (by setting all ACLs to DENY every request).

The changes made to the base environment are presented in Listing 5.5.

```
1 services:
2   broker:
3     ...
4   environment:
5     ...
6     KAFKA_AUTHORIZER_CLASS_NAME: org.apache.kafka.metadata.
authorizer.StandardAuthorizer
7     KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND: true
8   volumes:
9     - "./scripts:/scripts/"
```

Listing 5.5: Base environment changes for ACL bypass

`KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND` is set to `true` to account for cases where an ACL is not defined and as such the `StandardAuthorizer` would deny the corresponding operation.

During experimentation, an example topic was created and its deletion blacklisted by adding an ACL that denies topic deletions for every user principle. This means that this topic cannot be deleted by any normal user that connects to the Kafka cluster. However, if we think about the effect of deleting a topic (i.e., its data disappears from the system), we can realize that there are other ways of achieving the same effect, chief among them is changing the attribute `retention.ms`. This attribute controls the duration, in milliseconds, that inactive log segments should stay in the system before they are removed, if the retention policy is set to `delete`. By setting that value to 0 we are effectively telling Kafka that all logs should only be retained for a maximum duration of 0ms, that is, for no time at all. The net result is that, no matter the influx of messages arriving at a topic, it will always be empty, due to Kafka constantly cleaning up the topic's associated segments.

5.2.3 Summary

According to the preconditions set in Section 5.2.1, the developed Threat Model can be summarized as follows:

System Description An Event-Driven System designed around Apache Kafka.

Assumptions

- An attacker has infinite resources to conduct exploits;
- An attacker has access to the supply-chain of the system, being able to inject malicious code into legitimate packages;

Potential Threats

- Code injection via supply-chain attacks;
- Bypasses of non-exhaustive Access Control Lists;
- Deserialization issues caused by improper input validation

Remediation Actions

- Regular supply-chain audits; Fine-grained control to who can access the supply-chain of the system, attributing full accountability for introduced vulnerabilities;
- Regular audits of the system's Access Control Lists to prevent corner cases, even if they never happened; Implementation of more sophisticated authorization mechanisms;
- Proper input validation to all external input that the system receives; Maintenance of a list of common malicious payloads that might affect the system's dependencies in order to always stay up to date on recent security advisories;

Validation Validation of the Threat model is implicit in the sense that a successful exploitation attempt proves that a threat exists. By understanding the exploit itself, preventive or mitigative actions can be assessed and evaluated.

5.3 Policy Vulnerability Detection Tooling

Here, we introduce *Opfuzz*, a tool capable of consuming Rego policy files and, given some configuration as command-line arguments, fuzz the policy's input values to uncover any discrepancies between the expected policy contract and the actual policy implemented, without much manual intervention.

In Section 5.3.1, we dive deeper into the rationale behind building this tool and why it is important in the context of this work. Then, in Section 5.3.2 the actual components that make up the tool are succinctly described in terms of their functionality. Finally, a short summary of this subsection is given in Section 5.3.3.

5.3.1 Rationale

In the previous sections, we have introduced both a taxonomy of common vulnerability categories plaguing Event-Driven Systems (Section 5.1) and a Threat Model based on that same taxonomy detailing how to prevent and exploit those same vulnerabilities (Section 5.2). While these provide a great starting point for future research, they cover too much of the vulnerability/threat landscape of Event-Driven Systems, and developing a practical solution that addresses all of the mentioned issues is not only unfeasible but also out of scope for this dissertation. As such, only one of the many different aspects of Event-Driven Systems can and should be explored in this dissertation.

One crucial aspect of any system is authorization, a feature that dictates which entities can perform which actions against which resources, effectively posing itself as a first-line of defense against unwarranted access, and this is, as expected, a prevalent concern in Event-Driven Systems, such as those developed using a Microservices Architecture [3] due to the complex communication relationships formed between services and system components. There are several ways of achieving this authorization control: one of the most used methods of achieving authorization is with the use of **Access Control Lists**, or **ACLs**, which provide a direct mapping among entities, resources and actions. One specific implementation of this can be found in modern operating systems such as Linux or Windows [41], which map users/categories of users to resources using a bitmask or "special" number to identify those users' permissions (read, write, execute, etc.) to access certain system resources (directories or files). Another example implementation of ACLs can be found in Apache Kafka, where each ACL representation includes a *principal* (client or group of clients attempting to perform an operation, if wildcards are used), a given *resource* (or resources, if wildcards are used), an *operation* (read, write, describe, etc.) and a permission type (either *DENY* or *ALLOW* in Apache Kafka terminology) [70].

Another method that is seeing more traction in the Event-Driven ecosystem for enforcing authorization rules is **Policy enforcement**. This can be achieved through several mechanisms, one of them being the aforementioned ACLs, which can be thought of as a primitive and rather in-expressive type of policy. Another method of policy enforcement is the use of **Policy Engines**, specialized applications that present a more expressive way of defining policies and can respond to requests with a decision based on some arbitrary input. An example of this kind of application is **Open Policy Agent** (OPA) [52], an open-source policy evaluation engine developed under the *Cloud Native Computing Foundation* [10, 62] with the goal of decoupling application business logic from policy enforcement logic. Policies in OPA are written with the **Rego** [60, 62] policy language, a custom language based on Datalog designed specifically for writing policies that can leverage structured data such as JSON data. OPA itself is distributed as an executable which can run in interactive mode (as a CLI application) or in server mode (reachable through a network connection). Throughout the remainder of this document, when talking about OPA we are referring to the executable running in *server mode*. External data can be added to OPA for use inside a policy (such as organization-wide configurations that do not change between requests). OPA Policies are made up of *rules* that are evaluated against some input data and can reference other rules, thus forming a so called "rule graph".

Due to the growing adoption of OPA as a policy enforcement engine, the tool built as part of this dissertation's work will focus on exploring the landscape of Rego Policy-based authorization vulnerabilities. It does so by interpreting what kinds of input a policy consumes and generating various random values for said input. By doing this, we can, without much human intervention, generate a set of records that identify whether some data, derived from the policy's expected input values, is valid against the given policy, assuming the same environment (i.e., the global *data* object) is available to both OPA and Opfuzz.

5.3.2 System Components

In this section, we succinctly introduce the proper components that make up the architecture of *Opfuzz*, developed from the taxonomy described in Section 5.1 and from the Threat Model introduced in Section 5.2. Section 5.3.2.1 details the object model developed to represent Rego Policies in the context of this dissertation's work. Then, a method of generating random input for fuzzing said policies is described in Section 5.3.2.2. Finally, a succinct explanation of how all of these components come together to perform proper policy input fuzzing is given in Section 5.3.2.3.

5.3.2.1 Policy Object Model

Before attempting to perform value generation, any Rego file fed as input to the tool is converted to a custom object model of the Rego language, derived from the Rego language specification, henceforth called **Opom** (Open Policy Agent Object Model).

Opom includes "primitives" for representing *Policies*, *Rules*, which appear as top level members of a policy, *Queries*, which are rules inside a rule body, enclosed between brackets, *Expressions*, which can either evaluate to a boolean value and thus be used directly as if they were a query or evaluate to another type of value and can only appear as members of other expressions (thus meaning that expressions form themselves a tree-like structure), or values, such as *Scalars* or *References*.

There are various methods of generating a suitable model representation of a Rego policy, with the default implementation using ANTLR4 [59] as a parser generator, using a custom grammar based on the EBNF grammar published by Rego developers⁵. Manual creation of Policy objects is also possible, although extra care should be taken to keep vlaues and references consistent.

After the textual representation of the policy file is interpreted and converted into a suitable object model, the next step of the value generation pipeline is executed, described in the following section.

5.3.2.2 Policy Input Fuzzer

Once a suitable representation of the policy is derived from a Rego policy file, its input structure must be derived: this entails figuring out the shape of input data the policy can receive as well as the type of data associated with that shape, i.e., figuring out whether the policy can accept as input an object containing an arbitrary key mapped to a numerical value. In this way, Opif (Open Policy Agent Input Fuzzer) came to be.

Since, in Rego, rules form a "rule graph", it is important that our tool also contains such a representation. Even though we are not evaluating the actual policy, having knowledge of the precedence relationships between rules in a policy helps with value generation since some rules might be used by other rules, and knowing when to trigger value generation based on this relationships is paramount. The algorithm used for this step is presented in Algorithm 1. Note that each

⁵<https://www.openpolicyagent.org/docs/policy-reference#grammar>

different rule ref might have correspond to several rule definitions, each of them being a different vertex in the graph.

Once this rule graph is generated, it needs to be traversed to properly represent the input values that the policy expects. In order to respect the precedence relations between rules in the rule graph, traversal must happen in topological order. A custom implementation of the Visitor Pattern is used to process the graph's vertices and construct a suitable representation of the policy's input shape. The outcome of this step is a *Restriction* object. Currently implemented there are *BooleanRestriction* for generating boolean values, *StringRestriction* for string values, *NumberRestriction* for numerical values and *ObjectRestriction* for generating mappings from attributes to other restrictions. Each restriction object has some level of randomness associated, which will be responsible for the final value that is generated.

Once this step is done, value generation can begin according to the inferred input shape.

5.3.2.3 Policy Fuzzing Tool

During initialization, configuration values are read from the command line and properly parsed into their corresponding values. One of the most important values that needs to be parsed is the so called "stop condition": this is what controls when the value generation process stops. Currently there are 3 strategies implemented: *Default*, which never signals the value generator to stop, *AfterN* which takes as input an upper limit which, when reached, signals the value generator to stop generating values, *Custom*, which allows loading a custom implementation, which might suit more complex business needs.

Afterwards, Opfuzz starts communicating with an OPA instance, either running as a remote server or locally through the command line, sending values to it as they are generated and recording the response received. If a target for collecting validation errors is specified, these are appended when the validation decision indicates that the input is not valid. All of this data is collected in a validation report, which registers other pieces of information such as time taken and command-line arguments used to configure the tool, as well as statistics for a given execution. This report is then written to a file which can be analyzed by policy experts to validate that the obtained results either confirm the correct implementation of the policy or help uncover previously unknown cases which violate the policy's intended contract.

5.3.3 Summary

To sum up, the tool consists of three components, each responsible for a different step in the value generation process. First, any Rego files are parsed into a suitable Object Model representation. Next, this representations is traversed in order to generate a schema for the values the policy accepts as input. Lastly, values that respect this schema are generated and sent to OPA and the resulting validation decisions recorded for further manual analysis by policy experts.

Algorithm 1 Rule-Graph generation

```

1: procedure GENERATERULEGRAPH(policy)
2:    $graph \leftarrow \{\}$ 
3:    $inputNode \leftarrow \{ref = "input", rule = null\}$ 
4:    $dataNode \leftarrow \{ref = "data", rule = null\}$   $\triangleright$  input and data nodes are special Rego nodes
5:    $refToVertex \leftarrow \{\}$   $\triangleright$  used for retrieving vertices for dependencies
6:    $graph.ADD(inputNode)$ 
7:    $graph.ADD(dataNode)$ 
8:   for all  $ref \in policy.refs$  do  $\triangleright$  Add all graph vertices
9:      $rules \leftarrow policy.rules.GET(ref)$ 
10:    if  $\neg refToVertex.INCLUDES(ref)$  then
11:       $refToVertex[ref] \leftarrow \{\}$ 
12:    end if
13:    for all  $rule \in rules$  do
14:       $target \leftarrow rule.ref$ 
15:       $vertex \leftarrow \{ref = target, rule = rule\}$ 
16:       $refToVertex[target].APPEND(vertex)$ 
17:       $graph.ADDVERTEX(vertex)$ 
18:    end for
19:  end for
20:  for all  $ref \in refToVertex.keys$  do  $\triangleright$  Add all edges, i.e., dependency relations
21:     $vertices \leftarrow refToVertex.GET(ref)$ 
22:    for all  $vertex \in vertices$  do
23:      if  $\neg vertex.HASRULE()$  then
24:        continue
25:      end if
26:       $rule \leftarrow vertex.rule$ 
27:       $dependencyRefs \leftarrow rule.GETREFS()$ 
28:      if  $dependencyRefs = \emptyset$  then
29:        continue
30:      end if
31:      for all  $dependencyRef \in dependencyRefs$  do
32:        if  $dependencyRef.ISPREFIXEDBY("data")$  then
33:           $graph.ADDEDGE(vertex, dataNode)$ 
34:        else if  $dependencyRef.ISPREFIXEDBY("input")$  then
35:           $graph.ADDEDGE(vertex, inputNode)$ 
36:        else
37:           $dependencies \leftarrow refToVertex.GET(dependencyRef)$ 
38:          for all  $dependency \in dependencies$  do
39:             $graph.ADDEDGE(vertex, dependency)$ 
40:          end for
41:        end if
42:      end for
43:    end for
44:  end for
45: end procedure

```

5.4 Summary

In this chapter, a tripartite solution to the problems formalized in Chapter 4 is introduced.

First, a taxonomical categorization of vulnerabilities in Event-Driven Systems is introduced in Section 5.1. This taxonomy, introduced in Section 5.1.4, was formulated from a collection of 17 articles and publications from the gray literature, whose analysis methodology is described in Section 5.1.2. From this methodology, 4 categories could be identified: *Broken Access*, *Deserialization Issues*, *Policy Mismatch* and *Security Mismatch*. Finally, some limitations of the analysis process were detailed in Section 5.1.5.

Then, Section 5.2 formulates a Threat Model based on the previously defined taxonomy and following some guidelines laid by OWASP's Threat Modeling Project [54]. Several experiment environments were details in Sections 5.2.2 through 5.2.2.3, along with the exploitation attempts and potential consequences of successful attacks. Finally, all these findings are compiled into a Threat Model for Event-Driven Systems in Section 5.2.3.

Finally, in Section 5.3, we present a tool capable of "fuzzing" Rego policies with the goal of helping uncover unknown corner cases that violate the policy's contract. As explained in Section 5.3.3, it does this by first parsing a Rego policy file in order to construct a suitable schema for the kinds of data said policy accepts as input. Given this schema, the tool is capable of generating a large number of values that are then sent to OPA for validation, and the resulting validation decision is recorded in a specially formatted report.

Chapter 6

Validation

Contents

6.1	Context	52
6.2	Validation Goals	53
6.3	Results	54
6.3.1	Attack Experiments	54
6.3.2	Validation Report Analysis	55
6.4	Discussion	58
6.4.1	Threats to validity	59
6.5	Summary	60

Having introduced, in Chapter 5, a taxonomy of vulnerabilities in Event-Driven Systems, a Threat Model on those vulnerabilities and a tool capable of fuzzing Rego policies in order to find breaches in their contract, we can now validate that they confirm or deny the hypothesis introduced in Chapter 4, Section 4.2.

Firstly, some context about the validation methodology taken in this dissertation is given in Section 6.1. From this, validation goals were established in Section 6.2 and results presented in Section 6.3. A succinct discussion over the results found against this goals is presented in Section 6.4. Finally, a summary of the contents of this chapter is given in 6.5.

6.1 Context

There is no proper way to validate a document detailing a taxonomy on EDS vulnerabilities: it is a semi-static document, which can receive occasional updates to reflect the state of the industry at the time of it's (re)writing. Prime examples of this type of document are the aforementioned OWASP Top 10 Projects [53, 55]. Provided there are significant sources backing said taxonomy, all one can practically do is confirm whether the sources support the taxonomy or not. The same

rationale can be used when contemplating validating a corresponding Threat Model: all that can be done is confirm that the Threat Model correctly identifies a system's threat surface and correctly describes how attacks could be performed on that surface and how they should be prevented or mitigated. This problem is exacerbated if both of these documents are new, having no previous versions to build-up from, as is the case with the results discussed in this dissertation. One can, however, assert that the categorization made in the taxonomy is sound by attempting to replicate the threats identified in the Threat Model: a successful attack (granted it is performed in a safe and controlled environment with no malicious effect on real systems) further strengthens the claim that the categories in which the exploited vulnerabilities were placed are correctly defined.

However, validating that a tool designed around both the aforementioned taxonomy and threat model, like Opfuzz, does help uncover vulnerabilities that were previously unknown is indeed feasible. Even though Opfuzz does not encode any particular domain knowledge, and as such cannot itself confirm or deny that a policy vulnerability does exist, its generated reports can be useful for domain experts, namely Policy Experts who are in charge of developing and maintaining the given policies and as such are able to analyze the pairs of input and validation decision and confirm whether a given input should or should not have resulted in that decision being made. If there is any mismatch between the expected behavior of the policy and the actual experimental results, then we know that Opfuzz has indeed helped uncover some breaches of the policy's contract.

This way, the validation process by which the tripartite solution to the problem presented by this dissertation can confirm or deny the hypothesis introduced in Section 4.2 should consist of 2 separate parts:

- using the Threat Model experiments to validate the taxonomical categories;
- validating that the tool uncovers some breaches of a policy's contract;

6.2 Validation Goals

Given the contextualization and validation ideas presented in the previous section, we now present a formalization of the validation goals for this dissertation:

VG1: Successful experiments of some attacks support the introduced taxonomical categories

If a vulnerability is placed in the *Broken Access* category, there should be a way to exploit that vulnerability so that some form of unauthenticated or unauthorized (i.e., "broken" from the application's perspective) access happens. The same idea can be applied to every other category in the taxonomy presented in Chapter 5, Section 5.1. This validation process is implicit rather than explicit: it is the success of an exploit that determines the validity of both the categorization and the threat model. It should be noted that one exploit might relate to several different vulnerabilities, while one vulnerability might be triggered by several different exploits.

VG2: Breaches of a policy contract have been found via manual analysis of an Opfuzz-generated Report Using the reports generated by Opfuzz from an example policy, a domain expert should be able to manually analyze that report and uncover behavior that was previously unknown and that breaches the policies expected contract. This can happen in the form of input data not respecting its expected data type or the lack of some attribute not being accounted for.

6.3 Results

In this section, a description of the results obtained from the work developed throughout this dissertation are presented. In Section 6.3.1 the results obtained and discussed in Chapter 5, Section 5.2.1

6.3.1 Attack Experiments

As mentioned previously in Section 6.1, there is no easy way to validate the Threat Model, and thus the experiments, in a methodical way: by definition, an attack is valid if it is successful. This means that the validation process itself is implicit to the attacks.

Furthermore, in order to build the Threat Model, the experiments' results were already introduced and described. As such, in this section we present a brief summary of these findings:

- For experiment 1, a RCE vulnerability in Apache Avro was explored successfully, resulting in arbitrary code being executed. However, this exploit depends on a careful combination of several factors, which means it is practically infeasible to achieve in a real world setting;
- For experiment 2, a possible OutOfMemoryException caused by a dependency of Kafka's Java client library was the target of exploitation attempts. A successful exploit would mean a potential Denial-of-Service for the affected system. However, no attempt made at exploiting this vulnerability was successful;
- For experiment 3, we explored a potential code injection through a supply-chain attack on a Kafka-based system using custom policies for authorization of operations. Assuming an attacker as special access to the systems supply chain and can supply a malicious version of the policy authorization code, it could lead to arbitrary code being executed, potentially leading to a RCE.
- For experiment 4, Kafka-native authorization policies implemented as Access Control Lists are shown to be bypassable by understanding the semantics of certain operations and circumventing operation restrictions to disrupt the underlying system.

Further discussion of these results is given ahead in Section 6.4.

6.3.2 Validation Report Analysis

Several problems were uncovered by analysis of the report generated by Opfuzz. A portion of it can be seen in Appendix C. First of all, the most prevalent issue discovered was that the given policy was not taking into account missing input attributes, which resulted in incorrect evaluation of the policy's rules. An example of this is shown in Listing 6.1. Some text was redacted to exclude sensitive or confidential information. In this example, the attribute `topicName` is missing. Since this value is crucial for calculating the rest of the rules in the example policy, evaluation should have stopped as soon as it was detected that the value was missing. We know this is not the case since the policy attempted to compute a "service account" identifier from the `topicName` attribute, as seen in the first error message.

Also exemplified in Listing 6.1 is a payload where incorrectly typed attributes are not flagged. In this example, `confluent.tier.enable` refers to Confluent Platform's [13] configuration parameter with the same name.¹ As per the official documentation, this attribute can only assume boolean values, i.e., `true`, `false` and their text representations. As seen in this example, the policy itself does not catch the corner case where this attribute assumes an arbitrary value. While no major issues could occur (since it is assumed that this parameter only has an effect on validation if its value is one of the allowed values), this is still a corner case that, in a more critical situation, could potentially enable incorrectly validating some input. One possible cause that might have led to this case not being accounted for is that Command-Line Interfaces (CLIs) or Web UIs perform some kind of validation of their input data before sending that data to whatever system they are connected to. This validation might have taken into account the set of possible values for this attribute and, as such, it was never caught by the policy itself. This issue is the same for other attributes whose value is from an enumeration of values. An attacker with access to the payloads sent to the underlying system could exploit these corner cases to disrupt normal system operations.

```
1 {
2   "input": {
3     ...,
4     "configs": {
5       "confluent.tier.enable": "FW",
6       ...
7     },
8     ...
9   },
10  ...,
11  "valid": false,
12  "errors": [
```

¹<https://docs.confluent.io/platform/current/installation/configuration/topic-configs.html#confluent-tier-enable>

```

13     "Max number of topics per service account is not within ...
    limits.",
14     ...
15 ]
16 }

```

Listing 6.1: Example of payload that is missing some input or has incorrectly typed input

In Listing 6.2, the attribute `retention.ms` has a value that falls outside its range. This value, also present in Confluent Platform's configuration², only accepts values in the range $[-1, \infty)$, with -1 signaling that this property should be ignored. The example policy used, however, excludes -1 from its allowed values. Given this, we can conclude that this input also breaches the policy's contract. Despite the final validation decision coming out as invalid, it is so for reasons that are not caught by the policy itself: -2 should be an invalid value despite whatever additional restrictions a policy might impose (in this case, also restricting -1).

```

1 {
2   "input": {
3     ...,
4     "configs": {
5       ...,
6       "retention.ms": -2.0,
7       ...
8     },
9     ...
10  },
11  ...,
12  "valid": false,
13  "errors": [
14    ...,
15    "retention.ms is not within ... limits (== -1)."
16  ]
17 }

```

Listing 6.2: Example of payload that has values out of their accepted range

Finally, the last class of issues found with the policy itself is regarding the accepted property values for internal topics. These can be identified via their name, which start with an `"_"` character [35]. According to the findings from the validation report, in many cases where a given

²<https://docs.confluent.io/platform/current/installation/configuration/topic-configs.html#retention-ms>

payload should be considered invalid, the fact that the `topicName` variable holds the value `"_"` means that it is internal and, as such, bypasses several of the restrictions imposed by the rest of the policy. An example of such a case is present in Listing 6.3. In this example, the property `replicationFactor` is set to 4, a value which usually is not permitted (the correct value is 3). However, the fact that this input represents an internal topic means that any value could be associated with this topic, even negative ones.

```
1 {
2   "input":{
3     "replicationFactor":4.0,
4     "topicCountPerServiceAccount":102.0,
5     "topicName":"_",
6     "kafkaConfigs":{
7       "confluent.tier.enable":"false",
8       "log.segment.bytes":50000,
9       "log.roll.ms":1024
10    }
11  },
12  ...,
13  "valid":true
14 }
```

Listing 6.3: Internal topics may allow bypassing other restrictions

We can summarize the issues found in following list:

- The policy does not make sure that each attribute is of the correct data type;
- The policy does not validate that most attributes exist in the validation payload; This issue is more subtle since OPA implicitly validates attribute presence by assigning all references that are not in the input data a value of `undefined`, which is ignored in many operations. Even then, the example policy fails to respond correctly to the absence of certain critical input attributes.
- Some validations that are made by external tools (for example, complementary tools to interacting with Kafka) are not taken into account by the policy;
- Related to the second issue, the absence of the `topicName` attribute should immediately render the entire payload invalid. Instead, the policy still rejects these payloads but for different reasons;
- Internal topics seem to allow bypassing certain rules and restrictions of the policy;

6.4 Discussion

Revisiting the hypothesis that guided the work in this dissertation, *"By defining a taxonomy of vulnerabilities in Event-Driven Systems, (1) threat modeling for such systems can be simplified, and (2) the discovery of new vulnerabilities can be facilitated by providing a structured understanding of common attack vectors"*, we are made aware of the need of a taxonomical categorization of vulnerabilities in Event-Driven Systems as the basis from which the rest of the work needs to come. Introduced in Section 5.1, this taxonomy was developed from analysis of a *corpus* of on-line articles and blog posts related to Event-Driven Systems and some vulnerabilities and security issues plaguing these. Having this categorization defined, we need some way of validation that it makes sense and the several categories are well defined. To this end, a Threat Model for Kafka-based Event-Driven Systems was developed, detailing several threats, alongside exploits, and the possible mitigation steps to those threats.

Experimental results collected from building the Threat Model fell short from what was expected, with only 3 of the 4 attacks having been reported successful. Furthermore, one of these successful attacks was not a direct attack on a system with Apache Kafka deployed, but rather some component that is commonly used as a dependency either for Kafka itself or for systems that are usually built around Kafka. It can be argued then that the Threat Model does not in fact correctly model threats and mitigations to Event-Driven Systems based on Apache Kafka. This is recognized as a limitation of the work presented in this dissertation and further research should expand on the exploration done here.

Nonetheless, the exploits that were indeed successful are all centered around concepts native to Apache Kafka, so they provide some level of confidence not only about the (types of) vulnerabilities they were tested against but also that the Threat Model correctly identifies them as threats and, consequently, that the corresponding mitigation steps are a step in the right direction. We have thus the premise of our hypothesis, as well as the first hypothesized outcome: *"By defining a taxonomy of vulnerabilities in Event-Driven Systems, (1) threat modeling for such systems can be simplified, ..."*. We also can already answer Research Questions **RQ1**, *"What are the main security vulnerabilities present in systems employing an EDA and, more specifically, Apache Kafka?"*, and **RQ2**, *"What mechanisms exist for malicious threat actors to exploit those vulnerabilities against arbitrary systems?"*: the taxonomy presented in Section 5.1 identifies the main categories of vulnerabilities in Event-Driven Systems which in turn were formulated from a set of concrete reports of vulnerabilities and security issues (**RQ1**); through the threats and exploits described in the Threat Model introduced in Section 5.2, we can gain an understanding on how a threat actor would target and attempt to attack target systems based on the existence or not of those vulnerabilities in them (**RQ2**). We also see that Validation Goal **V1** is respected.

To validate the rest of the hypothesis, *"... and (2) the discovery of new vulnerabilities can be facilitated by providing a structured understanding of common attack vectors"*, Opfuzz was developed with the aim of helping uncover vulnerabilities in Rego policies. As mentioned previously, the threat landscape encompassed by both the developed taxonomy and Threat Model

is too broad to develop a tool that can reason about all of their aspects, and as such we decided to only focus on policy authorization via OPA. Designed to generate a report that would be later manually analyzed, it was discovered that a lot of the input values generated by Opfuzz resulted in incorrect validation decision by the Rego policy itself. We can then see that Validation Goal **V2** is also respected. Furthermore, we can now promptly answer Research Question **RQ3**, "*How can we provide a structured view of common attack vectors plaguing Event-Driven Systems?*": "a tool designed around uncovering policy vulnerabilities" has the benefit of helping uncover corner cases in a policy's contract that, if left unnoticed, could result in serious consequences should an attacker take advantage of them.

It is also worth mentioning that, as expected, there is a common theme surrounding all parts of the solution: in the taxonomy we identified *Policy Misconfigurations* as the third biggest category of vulnerabilities in Event-Driven Systems; in the Threat Model we saw how one could abuse this policy mechanism to run arbitrary code inside of and possibly gain access to a system; in Opfuzz, policies were the main subject of attention, and several misconfigurations were found in the example policy used for validating that the tools works as expected.

Given that all parts of the hypothesis were addressed, we can conclude that the hypothesis

"By defining a taxonomy of vulnerabilities in Event-Driven Systems, (1) threat modeling for such systems can be simplified, and (2) the discovery of new vulnerabilities can be facilitated by providing a structured understanding of common attack vectors."

is valid.

6.4.1 Threats to validity

As discussed in Section 5.1.5, the taxonomy of Event-Driven Systems' vulnerabilities was formulated from a corpus of 17 online articles, blog posts and security advisories. The sample size may be too small for any conclusions drawn to be considered valid. While care was taken to select a diverse set of examples from the gray literature, eventually search engine results started returning the same articles or different articles (i.e., published by different entities) having the same content. This places restrictions on the concrete number of distinct, *relevant*, publications that can be analyzed. Furthermore, the categories themselves were defined after analysis by a single individual. This of course means that any inherent bias in article analysis influenced the results obtained, threatening their validity.

The threat model, despite overlapping with many categories present in the vulnerability taxonomy, lacks extensive support, having been defined from only 4 exploits and threats. As said earlier, this begs the questions of whether this Threat Model is actually valid and whether it actually correctly represents the threat landscape surrounding Event-Driven Systems.

Opfuzz, despite having several limitations in its design, does not present any threats to the validity of the compiled results. No further threats could be identified.

6.5 Summary

Aiming to validate the hypothesis introduced in Chapter 4, Section 4.2, we begin this chapter by providing, in Section 6.1, some context on the problems that exist with having a properly structured validation methodology able to validate that the tripartite solution developed in this dissertation achieves its goal. However, it is recognized that a sort of *implicit* validation can be achieved between the vulnerability taxonomy and threat model: by successfully recreating the vulnerabilities associated with the taxonomy, albeit in a controlled environment, we can be more certain that the categorization was well made. Furthermore, it is discussed how Opfuzz-generated reports, while not providing themselves any recommendation of warning about potential vulnerabilities in a policy's contract, can be analyzed by a policy expert who can in turn assess whether some combination of input data and validation decision is incorrect for the intended implementation of the policy. From this discussion, two validation goals were defined and presented in Section 6.2, *Successful experiments of some attacks support the introduced taxonomical categories* and *Breaches of a policy contract have been found via manual analysis of an Opfuzz-generated Report*

Using these, in Section 6.3 we expose the results of conducting the validation process against the defined goals. On the one hand, the developed Threat Model includes details of some attacks that successfully exploited vulnerabilities present in categories of the taxonomy, thus helping strengthen the claim that the taxonomical division is correct. On the other hand, we showed how Opfuzz is capable of generating reports which helped uncover several unknown issues with an example Rego policy. These findings were then promptly discussed in Section 6.4, where we argued about the success of the developed components in validation the proposed hypothesis.

Chapter 7

Conclusion

Contents

7.1	Discussion	61
7.2	Contributions	63
7.3	Limitations	63
7.4	Future Work	64

This chapter presents the final remarks of this dissertation and concludes this document. In Section 7.1 we reiterate the background context and the motivation for developing this work, the problems found while reviewing the existing contributions made by the academia, the proposed solution to solve those problems and concluding discussions. The contributions mentioned in the previous section are summarized in Section 7.2, with limitations of the current work and future work presented, respectively, in sections 7.3 and 7.4.

7.1 Discussion

Software security has always been a cornerstone of software development, with companies often including in their development teams security experts whose sole responsibility is to audit, advise and help develop the security posture of a software product or even an entire system. From simple permission checks embedded in code to entire security solutions developed for the purpose of easing adoption of security practices by third-party agents, it is clear that software security is as, if not more, important to the SDLC than its other aspects. Furthermore, with the growing number of cybersecurity threats, as was the case in the UK in 2024 [61], it is paramount that our systems are secured against external malicious actors.

Several kinds of systems exist, with very different needs when it comes to securing them from external threats or preventing and mitigating known vulnerabilities: in a monolithic system, even though the attack surface of the system is smaller and all components are closely contained, often

leading to smaller mental effort in relating a vulnerability to its cause, fixing known security issues as a development step needs to be deeply integrated with all other aspects of the development process to ensure that no work stalls any other work. In contrast to this, systems which are (more) distributed in nature have the advantage of enjoying a (more) distributed development process, in which the security assurance process can be split per domain and as such be less demanding on the entire project. Prime examples of these systems are REST HTTP servers and systems developed under a Micro-Services architecture, respectively. Given their differences, it is expected that research on security in these types of systems focuses on different concerns of the systems and how they operate.

Despite their being an abundance of research on security-related topics, such as the various OWASP Foundation's Top 10 Projects [53, 55], based on surveys aimed at professional developers and personnel involved with the software development ecosystem, there is a lack of research around security and vulnerabilities in distributed systems, mainly Event-Driven Systems, as per the findings in Chapter 3. Evidently, there is some factor that is deterring any meaningful research on this topic from being published. This factor can either be just a general lack of interest, as can be perceived by the evolutionary analysis present in Chapter 3, Section 3.4, in which no trend can be perceived, or a coincidentally common phenomenon of research being done but remaining private due to business critical or sensitive information being present in the findings/discussion of said research. Whatever the reason is, it is clear that there is a large gap in academic literature regarding security vulnerabilities in Event-Driven Systems.

Aiming to fill a portion of that research gap, this dissertation introduces a *Taxonomy of Vulnerabilities in Event-Driven Systems*, a *Threat Model on Event-Driven Systems*, building up from the findings used to develop the taxonomy, and finally a Proof-of-Concept tool, *Opfuzz*, serves both as a fuzzer for policies written in Rego, the official policy language used by the Open Policy Agent software stack, and as an auxiliary step in manual policy auditing processes which aim to improve existing policies by uncovering previously unknown contract and security vulnerabilities. Firstly, since the academic literature was found lacking in terms of research regarding security vulnerabilities in Event-Driven Systems, we turned our efforts to the *gray literature* since this more closely reflected the point of view of industry practitioners who, after users/customers, are the most affected by successful exploits or attacks. A total of 17 web articles, obtained using the methodology described in Chapter 5, Section 5.1.2, were analyzed, distilled and compiled into 4 main categories according to various characteristics of the articles and the threats/vulnerabilities they go over such as what action triggers a given vulnerability and what prior system issues might allow the vulnerability under analysis to happen: *Broken Access*, *Deserialization Issues*, *Policy Misconfiguration* and *Security Misconfiguration*. Then, equipped with this taxonomical categorization, a Threat Model was developed aiming to detail how several of the previously introduced vulnerabilities can be exploited and potentially providing details on how those vulnerabilities might be mitigated/prevented. This Threat Model also serves as an implicit method of validation for the correctness of the categorization present in the taxonomy, i.e., that vulnerabilities are correctly grouped in their categories. 4 main threats associated with Deserialization issues and Broken Access were explored

and their mitigations laid down. Finally, and since the vulnerability landscape surrounding Event-Driven Systems is too vast to tackle all at once, we introduce *Opfuzz*, a Proof-of-Concept tool that specializes in a particular aspect of modern Event-Driven Systems: policy enforcement. Using Open Policy Language[52] and its policy language, Rego[60], *Opfuzz* is able to interpret a policy file and come up with a schema of the policy’s input shape. With this schema, it is able to generate random values, which are sent to OPA for validation. Afterwards, results are collected in a report. As detailed in the Chapter 6, this report is shared with a domain expert, in this case a policy contract expert, which confirmed that the report generated by *Opfuzz* led to the discovery of previously unknown breaches of the sample policy’s contract.

7.2 Contributions

The contributions of this work, described succinctly in the previous section, include the following items:

- A novel, taxonomical categorization of vulnerabilities in Event-Driven Systems, similar to established rankings in the industry [53, 55] and based on gray literature accounts from industry practitioners, describing 4 main categories of vulnerabilities in Event-Driven Systems: *Broken Access*, *Deserialization Issues*, *Policy Misconfiguration* and *Security Misconfiguration*;
- A newly created Threat Model that can be followed and studied to understand how to exploit the vulnerabilities found in the aforementioned taxonomy and how these might be prevented/mitigated;
- A specialized tool, *Opfuzz*, that can perform fuzzing tests against a commonly used Policy Evaluation Engine (OPA) and generate succinct reports from every execution, which have resulted in the discovery of previously unknown errors in an example policy;

7.3 Limitations

Despite achieving most if not all of the proposed goals in this dissertation, there are still some limitations present in the solutions introduced, some of which were introduced in Section 6.4.1 and which are further discussed in this section.

Firstly, regarding the developed Event-Driven Vulnerability Taxonomy, as introduced in Chapter 5, Section 5.1.2 and further elaborated upon in Section 5.1.5, in some articles the data interpreted from article analysis overlaps in one or more dimensions. This might be attributed to incorrect interpretation of the article’s text, which in turn could be attributed to some inherent bias introduced in the analysis process. This all stems from the fact that the analysis process was conducted by a single individual and not a group of individuals, which would help minimize the effects of such analysis biases. Moreover, due to time constraints, the number of analyzed

articles might be considered insufficient to draw significant conclusions regarding the categorization of vulnerabilities introduced. Since this dissertation work encompasses several other distinct, although connected, components, a more extensive article discovery step was considered out of scope. Further research could focus solely on the vulnerability taxonomy and expand on the work done in this dissertation.

Them, in the context of the developed Event-Driven Vulnerability Threat Model, it is recognized that the body of threats and exploits analyzed is considerably small. While a lot of them may overlap with the categories of the taxonomy, on which they were based, some fail to directly demonstrate an effective attack on Kafka-based systems. The problem also stems from the fact that this vulnerability exploration is very much an *ad hoc* effort. This begs the question about whether the Threat Model correctly models threats and their mitigations.

Lastly, considering the developed Policy Vulnerability Detection Tool, Opfuzz, the input schema generation method, despite covering a lot of what we assume are regular use cases of Rego policy definitions, is lacking in a lot of more complex situations. The fact that Opfuzz is not a policy engine means that we can't actually execute the policy to see the global effect each rule exerts on the final decision. This way, things like custom function definitions, iteration or associated attributes (i.e., cases where 2 different attributes should be validated simultaneously) are currently left unimplemented, affecting the final generated values in terms of coherency and variation. Also, the custom grammar defined for parsing Rego files does not exactly reflect the grammar published by the Rego authors, thus leading to flawed or outright incorrect parsing results and, as such, incorrect value generation. This stems from the fact that the published grammar is presented in EBNF format, which cannot be directly translated to the grammar format used by the tool. Care was taken to stay as faithful as possible to the original grammar but slight deviations were necessary in order to get a working system (such as removing *mutually left-recursive* grammar rules). Furthermore, there is no "shrinking" algorithm currently implemented in Opfuzz. This means that the generated report will more often than not contain redundant test cases, having different inputs which respect (or violate) the same set of properties in the same fashion and thus should be reduced to the minimal reproduction case. Implementing this requires more specialized knowledge of both the policy's input shape and of the various restrictions applied to that shape, which were not realized due to time constraints.

7.4 Future Work

From the examples of limitations of this work presented in the previous section, we can draw some conclusions regarding points of improvement for future work. First, the article discovery and analysis process should be repeated on a larger scale and with a larger group of researchers. This is meant to minimize the effect of analysis bias from researchers. A consensus protocol would need to be established to solve any divergences between article interpretations. A more extensive analysis process could also further strengthen the claims of the taxonomy, i.e., that the identified categories are indeed correct.

Then, more experiments based on the more extensive list of vulnerabilities taken from the previous point should be performed. This helps strengthen the findings of the threat model itself but also provides more data to be used as reference for the threats and mitigations found. The more threats, mitigations and overall experimentation is done, the closer to a *de facto* threat model of Event-Driven Systems we are considering contributions from more varied sources.

Also, the grammar used by Opfuzz should be further refined to cover all files that comply with the official Rego language grammar [60]. This would ensure that no corner cases would need to be encoded in the file parsing process. Care should also be taken to improve the tool itself to be more flexible and allow some level of policy execution in order to further expand its internal policy context when generating input for fuzzing.

Finally, a shrinking algorithm should be implemented, allowing the tool to generate only those inputs which are distinct between themselves but still generate different validation results. This process would need to take into account the input restrictions obtained from parsing the object model, itself parsed from the policy file. In doing so, generated validation reports could present less data that a manual operator needs to analyze while providing exactly the same value, since all relevant scenarios are included. Even though they are not directly related to input generation, some shrinking algorithms that could serve as a basis for this improvement were discussed in [44].

References

- [1] @micaturrin. *Interpreting Kafka Incidents—KAFKA-15096/15498/14800—michaelaturrin*. <https://medium.com/@michaelaturrin/interpreting-kafka-incidents-kafka-15096-15498-14800-a8f077d75fc6>.
- [2] Confluent Security Advisory. *CONFSA-2023-03: Confluent Platform Vulnerability in Schema Registry: Schema links can be misused to bypass Role Based Access Control (RBAC) based read restrictions on Schema Registry* — [support.confluent.io](https://support.confluent.io/hc/en-us/articles/15517380279060-CONFSA-2023-03-Confluent-Platform-Vulnerability-in-Schema-Registry-Schema-links-can-be-misused-to-bypass-Role-Based-Access-Control-RBAC-based-read-restrictions-on-Schema-Registry). <https://support.confluent.io/hc/en-us/articles/15517380279060-CONFSA-2023-03-Confluent-Platform-Vulnerability-in-Schema-Registry-Schema-links-can-be-misused-to-bypass-Role-Based-Access-Control-RBAC-based-read-restrictions-on-Schema-Registry>. 2023.
- [3] Murilo Góes de Almeida and Edna Dias Canedo. “Authentication and Authorization in Microservices Architecture: A Systematic Literature Review”. In: *Applied Sciences* 12.6 (2022). ISSN: 2076-3417. DOI: 10.3390/app12063023. URL: <https://www.mdpi.com/2076-3417/12/6/3023>.
- [4] AWS. *Amazon Managed Streaming for Apache Kafka*. URL: <https://aws.amazon.com/msk/>.
- [5] Philsy Baban. “Pre-processing and data validation in IoT data streams”. In: *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*. DEBS '20. event-place: Montreal, Quebec, Canada. New York, NY, USA: Association for Computing Machinery, 2020, pp. 226–229. ISBN: 978-1-4503-8028-7. DOI: 10.1145/3401025.3406443. URL: <https://doi.org/10.1145/3401025.3406443>.
- [6] Ken Barr. *Why You Need to Look Beyond Kafka for Operational Use Cases, Part 3: Securing the Eventing Platform* | Solace — [solace.com](https://solace.com/blog/beyond-kafka-3-securing-apache-kafka/). <https://solace.com/blog/beyond-kafka-3-securing-apache-kafka/>. 2020.
- [7] Broadcom. *RabbitMQ*. 2007. URL: <https://www.rabbitmq.com/>.
- [8] K. Mani Chandy. “Event Driven Architecture”. In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 1040–1044. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_570. URL: https://doi.org/10.1007/978-0-387-39940-9_570.
- [9] L. Chen. *oss-security - CVE-2025-27818: Apache Kafka: Possible RCE attack via SASL JAAS LdapLoginModule configuration* — [openwall.com](https://www.openwall.com/lists/oss-security/2025/06/09/2). <https://www.openwall.com/lists/oss-security/2025/06/09/2>. 2025.
- [10] *Cloud Native Computing Foundation* — [cncf.io](https://www.cncf.io/). <https://www.cncf.io/>.
- [11] Syn Cubes Community. *Pentesting Apache Kafka 101 : Top 5 Security Misconfigurations*. URL: <https://www.syncubes.com/pentesting-apache-kafka-101>.

- [12] Confluent. *Confluent Cloud*. URL: <https://www.confluent.io/confluent-cloud/>.
- [13] Confluent. *Confluent Platform*. URL: <https://www.confluent.io/product/confluent-platform/>.
- [14] CVE-2023-34455. <https://nvd.nist.gov/vuln/detail/cve-2023-34455#vulnConfigurationsArea>. June 2023.
- [15] CVE-2024-31141. <https://www.cve.org/CVERecord?id=CVE-2024-31141>. Nov. 2024.
- [16] CVE-2024-32030. <https://nvd.nist.gov/vuln/detail/CVE-2024-32030>. June 2024.
- [17] CVE-2024-47561. <https://nvd.nist.gov/vuln/detail/cve-2024-47561>. Mar. 2024.
- [18] CVE-2025-27817. <https://nvd.nist.gov/vuln/detail/CVE-2025-27817>. June 2025.
- [19] cve.org. <https://www.cve.org/>.
- [20] Prangshuman Das, Rodrigo Laigner, and Yongluan Zhou. “HawkEDA: a tool for quantifying data integrity violations in event-driven microservices”. In: *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*. DEBS ’21. event-place: Virtual Event, Italy. New York, NY, USA: Association for Computing Machinery, 2021, pp. 176–179. ISBN: 978-1-4503-8555-8. DOI: 10.1145/3465480.3467838. URL: <https://doi.org/10.1145/3465480.3467838>.
- [21] Pubali Datta et al. “Valve: Securing Function Workflows on Serverless Computing Platforms”. In: *Proceedings of The Web Conference 2020*. WWW ’20. event-place: Taipei, Taiwan. New York, NY, USA: Association for Computing Machinery, 2020, pp. 939–950. ISBN: 978-1-4503-7023-3. DOI: 10.1145/3366423.3380173. URL: <https://doi.org/10.1145/3366423.3380173>.
- [22] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. “A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations”. In: *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. AST ’20. event-place: Seoul, Republic of Korea. New York, NY, USA: Association for Computing Machinery, 2020, pp. 21–30. ISBN: 978-1-4503-7957-1. DOI: 10.1145/3387903.3389303. URL: <https://doi.org/10.1145/3387903.3389303>.
- [23] Bill Doerrfeld. *8 Tangible Ways — Including Least Privilege — to Secure Event-Driven Software Architecture* — [cloudwars.com](https://cloudwars.com/cybersecurity/how-to-enhance-cybersecurity-for-event-driven-software-architecture/). <https://cloudwars.com/cybersecurity/how-to-enhance-cybersecurity-for-event-driven-software-architecture/>. 2023.
- [24] Miguel Duarte et al. “Evaluation of IoT self-healing mechanisms using fault-injection in message brokers”. In: *Proceedings of the 4th International Workshop on Software Engineering Research and Practice for the IoT*. SERP4IoT ’22. event-place: Pittsburgh, Pennsylvania. New York, NY, USA: Association for Computing Machinery, 2023, pp. 9–16. ISBN: 978-1-4503-9332-4. DOI: 10.1145/3528227.3528567. URL: <https://doi.org/10.1145/3528227.3528567>.
- [25] Eswar. *Apache Kafka Flaw Let Attackers Gain Access To Sensitive Data* — [cybersecuritynews.com](https://cybersecuritynews.com/apache-kafka-security-flaw/). <https://cybersecuritynews.com/apache-kafka-security-flaw/>. 2024.

- [26] Roy T. Fielding and Richard N. Taylor. “Principled design of the modern Web architecture”. In: *ACM Trans. Internet Technol.* 2.2 (May 2002), pp. 115–150. ISSN: 1533-5399. DOI: [10.1145/514183.514185](https://doi.org/10.1145/514183.514185). URL: <https://doi.org/10.1145/514183.514185>. (accessed: 05.10.2024).
- [27] Apache Software Foundation. *Apache Kafka*. URL: <https://kafka.apache.org/>.
- [28] Apache Software Foundation. *Apache Kafka - Powered By*. URL: <https://kafka.apache.org/powered-by>.
- [29] The Apache Software Foundation. *Apache Avro — avro.apache.org*. <https://avro.apache.org/>.
- [30] Apostolos Giannakidis. *Reducing the Insecure Deserialization Risk — cyberdefensemagazine.com*. <https://www.cyberdefensemagazine.com/reducing-the-insecure/>. 2019.
- [31] Google. *Google Cloud Managed Service for Apache Kafka*. URL: <https://cloud.google.com/products/managed-service-for-apache-kafka?hl=en>.
- [32] Steffen Herbold, Jens Grabowski, and Stephan Waack. “A Model for Usage-Based Testing of Event-Driven Software”. In: *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion*. 2011, pp. 172–178. DOI: [10.1109/SSIRI-C.2011.32](https://doi.org/10.1109/SSIRI-C.2011.32).
- [33] B. Huang et al. “BPS: A reliable and efficient pub/sub communication model with blockchain-enhanced paradigm in multi-tenant edge cloud”. English. In: *Journal of Parallel and Distributed Computing* 143 (2020). Publisher: Academic Press Inc., pp. 167–178. ISSN: 07437315 (ISSN). DOI: [10.1016/j.jpdc.2020.05.005](https://doi.org/10.1016/j.jpdc.2020.05.005). URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85085552783&doi=10.1016%2fj.jpdc.2020.05.005&partnerID=40&md5=c7e933e6987ef1492702a7827a30ee34>.
- [34] IBM. *Security Bulletin: Security vulnerability in Apache Kafka clients affects IBM Business Automation Workflow Case Event Emitters - CVE-2024-31141 — ibm.com*. <https://www.ibm.com/support/pages/security-bulletin-security-vulnerability-apache-kafka-clients-affects-ibm-business-automation-workflow-case-event-emitters-cve-2024-31141-0>. 2024.
- [35] Confluent Inc. *Manage Topics Using Control Center for Confluent Platform | Confluent Documentation — docs.confluent.io*. <https://docs.confluent.io/control-center/current/topics/overview.html>.
- [36] Keshav Kaushik, Alpna Dass, and Ayush Dhankhar. “An approach for exploiting and mitigating Log4J using Log4Shell vulnerability”. In: *2022 3rd International Conference on Computation, Automation and Knowledge Management (ICCAKM)*. Nov. 2022, pp. 1–6. DOI: [10.1109/ICCAKM54721.2022.9990554](https://doi.org/10.1109/ICCAKM54721.2022.9990554).
- [37] Simon Kemp. *Digital 2022: Digital Adoption Doubled Over the Past Decade — DataReportal – Global Digital Insights — datareportal.com*. <https://datareportal.com/reports/digital-2022-digital-adoption-doubled-over-the-past-decade>.

- [38] Umma Khatuna Jannat, M.Mohan Kumar, and Syed Arif Islam. “Enhancing Cybersecurity Resilience: Real-time Ransomware Detection using AES Algorithm on Kafka Stream”. In: *2023 7th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. Nov. 2023, pp. 1818–1826. DOI: [10.1109/ICECA58529.2023.10394789](https://doi.org/10.1109/ICECA58529.2023.10394789).
- [39] Jay Kreps, Neha Narkhede, Jun Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. Vol. 11. 2011. Athens, Greece. 2011, pp. 1–7.
- [40] Bruno Kreyssig and Alexandre Bartel. “Analyzing Prerequisites of known Deserialization Vulnerabilities on Java Applications”. In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE '24. event-place: Salerno, Italy. New York, NY, USA: Association for Computing Machinery, 2024, pp. 28–37. ISBN: 9798400717017. DOI: [10.1145/3661167.3661176](https://doi.org/10.1145/3661167.3661176). URL: <https://doi.org/10.1145/3661167.3661176>.
- [41] Aneesh Kumar KV, Andreas Grünbacher, and Greg Banks. “Implementing an advanced access control model on Linux”. In: *Linux Symposium*. Citeseer. 2010, p. 19.
- [42] L. Florin Ilca and T. Balan. “Windows Communication Foundation Penetration Testing Methodology”. In: *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*. Journal Abbreviation: 2021 16th International Conference on Engineering of Modern Electric Systems (EMES). June 2021, pp. 1–4. DOI: [10.1109/EMES52337.2021.9484145](https://doi.org/10.1109/EMES52337.2021.9484145).
- [43] Yi Liu et al. “Morest: model-based RESTful API testing with execution feedback”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1406–1417. ISBN: 9781450392211. DOI: [10.1145/3510003.3510133](https://doi.org/10.1145/3510003.3510133). URL: <https://doi.org/10.1145/3510003.3510133>. (accessed: 05.10.2024).
- [44] Lars Meijer. “Test case shrinking for Model Based Testing on Symbolic Transition Systems”. MA thesis. University of Twente, 2021.
- [45] Atif M. Memon. “Developing Testing Techniques for Event-Driven Pervasive Computing Applications”. In: *Proceedings of The OOPSLA 2004 workshop on Building Software for Pervasive Computing (BSPC 2004)*. Oct. 2004.
- [46] Victor Moreno. *Lessons Learned From The Trenches Developing Event-Driven Systems* — [blog.stackademic.com](https://blog.stackademic.com/lessons-learned-from-the-trenches-developing-event-driven-systems-350272af477a). <https://blog.stackademic.com/lessons-learned-from-the-trenches-developing-event-driven-systems-350272af477a>. 2024.
- [47] Rabbit MQ. *RabbitMQ | RabbitMQ Queue Policy Mismatch* — [drdroid.io](https://drdroid.io/stack-diagnosis/rabbitmq-queue-policy-mismatch). <https://drdroid.io/stack-diagnosis/rabbitmq-queue-policy-mismatch>.
- [48] Dotan Nahum. *Misconfigured Kafdrop Puts Companies' Apache Kafka Completely Exposed - Spectral* — [spectralops.io](https://spectralops.io/blog/misconfigured-kafdrop-puts-companies-apache-kafka-completely-exposed/). <https://spectralops.io/blog/misconfigured-kafdrop-puts-companies-apache-kafka-completely-exposed/>. 2021.
- [49] Ravie Lakshmanan — The Hacker News. *Critical Apache Avro SDK Flaw Allows Remote Code Execution in Java Applications* — [thehackernews.com](https://thehackernews.com/2024/10/critical-apache-avro-sdk-flaw-allows.html). <https://thehackernews.com/2024/10/critical-apache-avro-sdk-flaw-allows.html>. 2024.

- [50] Henrik Nielsen, Roy T. Fielding, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. May 1996. DOI: 10.17487/RFC1945. URL: <https://www.rfc-editor.org/info/rfc1945>. (accessed: 05.10.2024).
- [51] *NVD - Categories* — [nvd.nist.gov](https://nvd.nist.gov/vuln/categories). <https://nvd.nist.gov/vuln/categories>.
- [52] *Open Policy Agent - Homepage* | *Open Policy Agent* — [openpolicyagent.org](https://www.openpolicyagent.org/). <https://www.openpolicyagent.org/>.
- [53] OWASP. *OWASP Cloud-Native Application Security Top 10* | *OWASP Foundation* — [owasp.org](https://owasp.org/www-project-cloud-native-application-security-top-10/#). <https://owasp.org/www-project-cloud-native-application-security-top-10/#>.
- [54] OWASP. *OWASP Threat Modeling Project* | *OWASP Foundation* — [owasp.org](https://owasp.org/www-project-threat-model). <https://owasp.org/www-project-threat-model>.
- [55] OWASP. *OWASP Top Ten* | *OWASP Foundation* — [owasp.org](https://owasp.org/www-project-top-ten/). <https://owasp.org/www-project-top-ten/>.
- [56] OWASP. *OWASP API Security Project* | *OWASP Foundation* — [owasp.org](https://owasp.org/www-project-api-security/). <https://owasp.org/www-project-api-security/>. 2023.
- [57] OWASP. *OWASP Mobile Top 10* | *OWASP Foundation* — [owasp.org](https://owasp.org/www-project-mobile-top-10/). <https://owasp.org/www-project-mobile-top-10/>.
- [58] Lev Pachmanov. *Critical Alert: CVE-2025-27817 in Apache Kafka Client* | *Seal Security* — [seal.security](https://www.seal.security/blog/cve-2025-27817-why-the-kafka-oauth-vulnerability-isnt-really-fixed-in-3-9-1). <https://www.seal.security/blog/cve-2025-27817-why-the-kafka-oauth-vulnerability-isnt-really-fixed-in-3-9-1>. 2025.
- [59] Terrence Parr. *ANTLR* — [antlr.org](https://www.antlr.org/). <https://www.antlr.org/>.
- [60] *Policy Language* | *Open Policy Agent* — [openpolicyagent.org](https://www.openpolicyagent.org/docs/policy-language). <https://www.openpolicyagent.org/docs/policy-language>.
- [61] Saman Rizvi and Eleanor Fordham. *Cyber security breaches survey 2025*. Apr. 2025. URL: <https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2025/cyber-security-breaches-survey-2025>.
- [62] Niklas Sanger and Sebastian Abeck. “User Authorization in Microservice-Based Applications”. In: *Software* 2.3 (2023), pp. 400–426. ISSN: 2674-113X. DOI: 10.3390/software2030019. URL: <https://www.mdpi.com/2674-113X/2/3/19>.
- [63] Arooba Shahoor, Jooyong Yi, and Dongsun Kim. “Preserving Reactiveness: Understanding and Improving the Debugging Practice of Blocking-Call Bugs”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. event-place: Vienna, Austria. New York, NY, USA: Association for Computing Machinery, 2024, pp. 768–780. ISBN: 9798400706127. DOI: 10.1145/3650212.3680319. URL: <https://doi.org/10.1145/3650212.3680319>.
- [64] Shoreline. *Kafka Unauthorized Access* | *Shoreline Runbooks* — [shoreline.io](https://www.shoreline.io/runbooks/kafka/kafka-unauthorized-access-due-to-kafka-acl-misconfiguration). <https://www.shoreline.io/runbooks/kafka/kafka-unauthorized-access-due-to-kafka-acl-misconfiguration>.
- [65] Navdeep Sidhu. *Common Kafka Security Misconfigurations and How to Avoid Them* — [meshiq.com](https://www.meshiq.com/common-kafka-security-misconfigurations-and-how-to-avoid-them/). <https://www.meshiq.com/common-kafka-security-misconfigurations-and-how-to-avoid-them/>. 2024.
- [66] J Simpson. *20 Impressive API Economy Statistics*. 2022. URL: <https://nordicapis.com/20-impressive-api-economy-statistics/>. (accessed: 09.10.2024).

- [67] J. Singh and N.K. Chaudhary. “Rest security framework for event streaming bus architecture”. English. In: *International Journal of Information Technology (Singapore)* 16.5 (2024). Publisher: Springer Science and Business Media B.V., pp. 3033–3047. ISSN: 25112104 (ISSN). DOI: [10 . 1007 / s41870 - 024 - 01836 - 8](https://doi.org/10.1007/s41870-024-01836-8). URL: [https : / / www . scopus . com / inward / record . uri ? eid = 2 - s2 . 0 - 85190361741 & doi = 10 . 1007 % 2fs41870 - 024 - 01836 - 8 & partnerID = 40 & md5 = 3f776fd2e1b74facfef2c378578cb44d](https://www.scopus.com/inward/record.uri?eid=2-s2.0-85190361741&doi=10.1007%2fs41870-024-01836-8&partnerID=40&md5=3f776fd2e1b74facfef2c378578cb44d).
- [68] Prashast Srivastava et al. “Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. event-place: San Francisco, CA, USA. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1586–1597. ISBN: 9798400703270. DOI: [10 . 1145 / 3611643 . 3616313](https://doi.org/10.1145/3611643.3616313). URL: [https : / / doi . org / 10 . 1145 / 3611643 . 3616313](https://doi.org/10.1145/3611643.3616313).
- [69] Muhammad Waseem et al. “On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study”. In: *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*. EASE '21. event-place: Trondheim, Norway. New York, NY, USA: Association for Computing Machinery, 2021, pp. 201–210. ISBN: 978-1-4503-9053-8. DOI: [10 . 1145 / 3463274 . 3463337](https://doi.org/10.1145/3463274.3463337). URL: [https : / / doi . org / 10 . 1145 / 3463274 . 3463337](https://doi.org/10.1145/3463274.3463337).
- [70] Purshotam Singh Yadav. “Advanced Authentication and Authorization Mechanisms in Apache Kafka: Enhancing Security for High-Volume Data Processing Environments”. In: *Journal of Engineering and Applied Sciences Technology*. SRC/JEAST-E110. DOI: [doi . org / 10.47363 / JEAST / 2024 \(6 \) E110](https://doi.org/10.47363/JEAST/2024(6)E110) 6.8 (2024), pp. 2–6.
- [71] Uwe Zdun et al. “Microservice Security Metrics for Secure Communication, Identity Management, and Observability”. In: *ACM Trans. Softw. Eng. Methodol.* 32.1 (Feb. 2023). Place: New York, NY, USA Publisher: Association for Computing Machinery. ISSN: 1049-331X. DOI: [10 . 1145 / 3532183](https://doi.org/10.1145/3532183). URL: [https : / / doi . org / 10 . 1145 / 3532183](https://doi.org/10.1145/3532183).
- [72] Man Zhang et al. “Seeding and Mocking in White-Box Fuzzing Enterprise RPC APIs: An Industrial Case Study”. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. event-place: Sacramento, CA, USA. New York, NY, USA: Association for Computing Machinery, 2024, pp. 2024–2034. ISBN: 9798400712487. DOI: [10 . 1145 / 3691620 . 3695265](https://doi.org/10.1145/3691620.3695265). URL: [https : / / doi . org / 10 . 1145 / 3691620 . 3695265](https://doi.org/10.1145/3691620.3695265).
- [73] Xiang Zhou et al. “Latent error prediction and fault localization for microservice applications by learning from system trace logs”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. event-place: Tallinn, Estonia. New York, NY, USA: Association for Computing Machinery, 2019, pp. 683–694. ISBN: 978-1-4503-5572-8. DOI: [10 . 1145 / 3338906 . 3338961](https://doi.org/10.1145/3338906.3338961). URL: [https : / / doi . org / 10 . 1145 / 3338906 . 3338961](https://doi.org/10.1145/3338906.3338961).

Appendix A

Event-Driven Systems' Vulnerability Taxonomy

Search Keywords

Here we present a more extensive list than the one introduced in Chapter 5, Section 5.1.2.

- Rabbit MQ
- Policy Mismatch
- Kafka
- Confluent
- Confluent Platform
- Event Driven
- Vulnerability
- Serialization
- Deserialization
- Issue
- Avro
- RCE
- Broken Access
- Security

Analyzed articles

The article analysis process was compiled into a spreadsheet, available at

<https://docs.google.com/spreadsheets/d/1SIu7HxY-8y7wgWdLAMqCI1NzF5d4fIpQSJ8PsmZMQ7E/edit?usp=sharing>.

Appendix B

Event-Driven Systems' Threat Model

B.1 Base Test Environment Configuration

```
1 ---
2 services:
3   broker:
4     image: confluentinc/cp-server:7.7.1
5     hostname: broker
6     container_name: broker
7     restart: always
8     ports:
9       - "9092:9092"
10      - "9101:9101"
11     environment:
12       KAFKA_NODE_ID: 1
13       KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: 'CONTROLLER:PLAINTEXT
14       ,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT'
15       KAFKA_ADVERTISED_LISTENERS: 'PLAINTEXT://broker:29092,
16       PLAINTEXT_HOST://localhost:9092'
17       KAFKA_METRIC_REPORTERS: io.confluent.metrics.reporter.
18       ConfluentMetricsReporter
19       KAFKA_CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: 'broker
20       :9092'
21       KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
22       KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
23       KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
24       KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
25       KAFKA_JMX_PORT: 9101
```

```

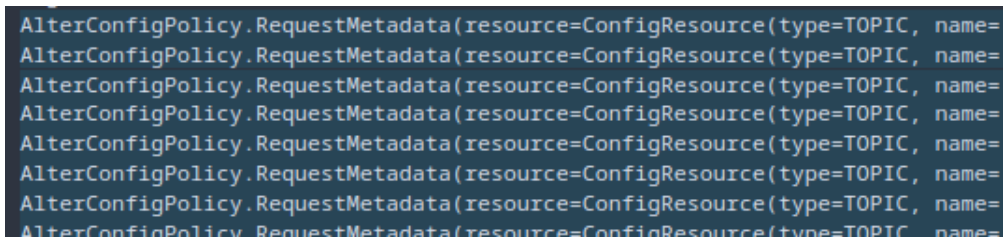
22     KAFKA_JMX_HOSTNAME: localhost
23     KAFKA_PROCESS_ROLES: 'broker,controller'
24     KAFKA_CONTROLLER_QUORUM_VOTERS: '1@broker:29093'
25     KAFKA_LISTENERS: 'PLAINTEXT://broker:29092,CONTROLLER://
broker:29093,PLAINTEXT_HOST://0.0.0.0:9092'
26     KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
27     KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
28     KAFKA_CONFLUENT_SCHEMA_REGISTRY_URL: http://schema-registry
:8081
29     KAFKA_CONFLUENT_METRICS_REPORTER_TOPIC_REPLICAS: 1
30     KAFKA_CONFLUENT_METRICS_ENABLE: 'false'
31     KAFKA_CONFLUENT_SUPPORT_CUSTOMER_ID: 'anonymous'
32     CLUSTER_ID: 'MkU3OEVBNTcwNTJENDM21k'
33     volumes:
34     - './scripts/:/scripts/'
35
36     schema-registry:
37     image: confluentinc/cp-schema-registry:7.7.1
38     hostname: schema-registry
39     container_name: schema-registry
40     depends_on:
41     - broker
42     restart: always
43     ports:
44     - "8081:8081"
45     environment:
46     SCHEMA_REGISTRY_HOST_NAME: schema-registry
47     SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: 'broker:29092
',
48     SCHEMA_REGISTRY_LISTENERS: http://0.0.0.0:8081
49
50     control-center:
51     image: confluentinc/cp-enterprise-control-center:7.7.1
52     hostname: control-center
53     container_name: control-center
54     depends_on:
55     - broker
56     - schema-registry
57     restart: always
58     ports:

```

```
59     - "9021:9021"
60     environment:
61         CONTROL_CENTER_BOOTSTRAP_SERVERS: 'broker:29092'
62         CONTROL_CENTER_SCHEMA_REGISTRY_URL: "http://schema-registry
:8081"
63         CONTROL_CENTER_REPLICATION_FACTOR: 1
64         CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS: 1
65         CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS: 1
66         CONFLUENT_METRICS_TOPIC_REPLICATION: 1
67         PORT: 9021
68
69     rest-proxy:
70         image: confluentinc/cp-kafka-rest:7.7.1
71         depends_on:
72             - broker
73             - schema-registry
74         restart: always
75         ports:
76             - 8082:8082
77         hostname: rest-proxy
78         container_name: rest-proxy
79         environment:
80             KAFKA_REST_HOST_NAME: rest-proxy
81             KAFKA_REST_BOOTSTRAP_SERVERS: 'broker:29092'
82             KAFKA_REST_LISTENERS: "http://0.0.0.0:8082"
83             KAFKA_REST_SCHEMA_REGISTRY_URL: 'http://schema-registry
:8081
```

Listing B.1: Base Experiment Environment

Bad Policy Authorizer sample log output

A screenshot of a log file showing a series of repeated log entries. Each entry is a single line of text: `AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=`. The text is white on a dark background, and the lines are stacked vertically, with the last line being partially cut off at the end.

```
AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=  
AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=  
AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=  
AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=  
AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=  
AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=  
AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=  
AlterConfigPolicy.RequestMetadata(resource=ConfigResource(type=TOPIC, name=
```

Figure B.1: Screenshot of example logs from potentially malicious policy authorizer

Appendix C

Opfuzz

Example Validation Report

```
{
  "beginDate": 1750278267951,
  "endDate": 1750278269806,
  "config": {
    "opaServerAddress": "http://localhost:8181",
    "policy": {
      "filePath": "./src/main/resources/custom2.rego",
      "target": "valid",
      "errorTarget": "violation"
    },
    "data": {
      "filePath": "./src/main/resources/data.yaml",
      "value": null
    },
    "fuzzingProbability": 0.5,
    "stopCondition": "after:100"
  },
  "stats": {
    "numberOfDecisions": 100,
    "results": {
      "false": 78,
      "true": 22
    }
  },
  "fuzzingDecisions": [
    {
      "input": {}
      "replicationFactor": 4.0,
      "configs": {
        "compression.type": "B",
        "cleanup.policy": "delete",
        "max.message.bytes": 1048588.0,
        "retention.ms": 15552000000.0,
        "segment.ms": 1024,
        "segment.bytes": 500000,
        "unclean.leader.election.enable": "true",
        "retention.bytes": 1.0
      },
      "numPartitions": 7.0
    },
    {
      "recordedAt": 1750278268330,
      "valid": false,
      "errors": [
        "Max number of partitions per topic is not within ... limits",
        "Replication factor has to be 3",
        "not allowed to override compression.type at topic level",
        "not allowed to override segment.bytes",
        "not allowed to override segment.ms",
        "not allowed to override unclean.leader.election.enable",
        "retention.bytes value must be -1 (unlimited)."
      ]
    }
  ]
}
```

Figure C.1: Screenshot of example validation report

Example Tool Configuration

```
1 --address "http://localhost:8181"
2 --policy-path "./src/main/resources/custom2.rego"
3 --data-path "./src/main/resources/data.yaml"
4 --target valid
5 --error-target violation
6 --fuzzing-probability 0.5
7 --output-file-type "json"
8 --stop-condition "after:100"
```

Listing C.1: Example tool configuration