future internet

Article

KubeHound: Detecting Microservices' Security Smells in Kubernetes Deployments

Giorgio Dell'Immagine, Jacopo Soldani * and Antonio Brogi

Department of Computer Science, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy
* Correspondence: jacopo.soldani@unipi.it

Abstract: As microservice-based architectures are increasingly adopted, microservices security has become a crucial aspect to consider for IT businesses. Starting from a set of "security smells" for microservice applications that were recently proposed in the literature, we enable the automatic detection of such smells in microservice applications deployed with Kubernetes. We first introduce possible analysis techniques to automatically detect security smells in Kubernetes-deployed microservices. We then demonstrate the practical applicability of the proposed techniques by introducing KubeHound, an extensible prototype tool for automatically detecting security smells in microservice applications, and which already features a selected subset of the discussed analyses. We finally show that KubeHound can effectively detect instances of security smells in microservice applications by means of controlled experiments and by applying it to existing, third-party applications.

Keywords: microservices; smell detection; security; Kubernetes

Citation: Dell'Immagine, G.; Soldani, J.; Brogi, A. KubeHound: Detecting Microservices' Security Smells in Kubernetes Deployments. *Future Internet* **2023**, *15*, 228. https://doi.org/10.3390/fi15070228

Academic Editors: Weizhi Meng and Christian D. Jensen

Received: 29 May 2023
Revised: 15 June 2023
Accepted: 23 June 2023
Published: 26 June 2023

1. Introduction

Microservices are being increasingly adopted by the IT industry, with companies such as Amazon, Netflix, and Spotify already delivering their core businesses through microservices [1,2]. This is because microservices enable obtaining so-called *cloud-native* applications, namely, applications fully exploiting the potentials of cloud computing, and since microservices smoothly integrate with the widespread DevOps practices [3].

Microservice architectures can be seen as peculiar extensions of service-oriented architectures (SOAs), characterised by an extended set of design principles, e.g., designing microservices around business concepts, making them independently deployable and highly observable, and isolating their possible failures [4]. Microservices hence inherit the traditional security concerns for SOA, whilst also raising new security challenges [5]. For instance, compared to SOA, microservices increase the surface prone to security attacks and raise the need to establish trust among the microservices forming an application and manage distributed secrets [6]. Furthermore, microservices interactions should be thoroughly secured, not only when interacting with external clients, but also whenever a microservice communicates with another microservice of the same application [7].

In this perspective, Ponce et al. [1] recently elicited the smells and refactorings for microservice security. They defined the "security smells" for microservices as possible symptoms of bad (though often unintentional) design decisions, which can negatively affect the security of a microservice application. Examples of security smells are non-secured service communications and unauthenticated traffic, which occur when the interactions among an application's microservices are not encrypted nor authenticated. Security smells can be resolved by refactoring an application or the microservices therein. For instance, the use of Mutual TLS and OpenID Connect enables the resolution of the above-mentioned smells [1]. However, automatically detecting the security smells affecting an existing microservice application is still an open problem [8].

In this perspective, the objective of this work is to demonstrate that instances of the security smells by Ponce et al. [1] can be automatically detected in microservice applications, while also providing an extensible tool for supporting this task. More precisely, the main contributions of this work are the following:

(i) We demonstrate that security smell detection can be automated by proposing a set of techniques enabling the detection of such smells in microservice applications deployed with Kubernetes, the de facto standard for orchestrating microservices [5].

(ii) We introduce KUBEHOUND, an open-source security-smell-detection tool that already features the implementation of a selected subset of the analyses in (i). KUBEHOUND is extensible and modular, enabling users to specify custom plugins that implement additional analyses to be applied.

(iii) We assess our approach in practice by running KUBEHOUND in controlled experiments with a "mock" microservice application and by applying it in two case studies based on existing, third-party applications.

It is worth noting that KUBEHOUND follows a "best effort" approach, meaning that it runs a smell-detection technique only if all the resources it needs are available. In other words, KUBEHOUND performs the largest possible subset of smell-detection techniques it can, given the inputs provided by an end-user. This enables end-users to provide only what they wish to be analysed, e.g., also enabling KUBEHOUND to work if the source code of some microservices is not available or if users do not wish KUBEHOUND to interact with a running instance of the target microservice application.

The rest of this article is organised as follows. Section 2 provides the necessary background on microservice security smells [1]. Section 3 discusses the relevant related work. Section 4 outlines possible analysis methods to automatically detect security smells. Section 5 describes the architecture and implementation of KUBEHOUND, together with the implementation of six plugins that perform smell detection. Finally, Sections 6 and 7 present the assessment of KUBEHOUND and draw some concluding remarks, respectively.

2. Background

We hereafter recall the security smells for microservices proposed by Ponce et al. [1], which they organised in the taxonomy in Figure 1. We then show how to detect instances of each of such smell in Kubernetes-deployed microservices in Section 4.

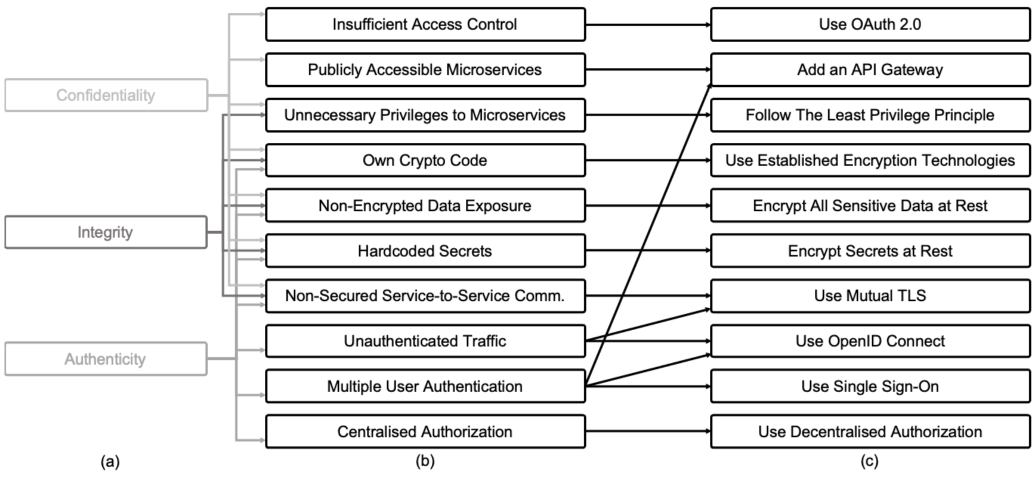
Confidentiality
Integrity
Authenticity
Insufficient Access Control
Publicly Accessible Microservices
Unnecessary Privileges to Microservices
Own Crypto Code
Non-Encrypted Data Exposure
Hardcoded Secrets
Non-Secured Service-to-Service Comm.
Unauthenticated Traffic
Multiple User Authentication
Centralised Authorization
Use OAuth 2.0
Add an API Gateway
Follow The Least Privilege Principle
Use Established Encryption Technologies
Encrypt All Sensitive Data at Rest
Encrypt Secrets at Rest
Use Mutual TLS
Use OpenID Connect
Use Single Sign-On
Use Decentralised Authorization
(a)
(b)
(c)

Figure 1. Taxonomy showing the (**a**) affected security properties, (**b**) security smells, and (**c**) refactorings for microservices [1].

Insufficient Access Control. This smell occurs when a microservice does not enforce access control. The lack of access control may indeed compromise the *Confidentiality* of a microservice, as attackers can trick it and access data/functions that should not be accessible to them. The possible effects of *Insufficient Access Control* can be mitigated by applying the *Use OAuth 2.0* refactoring, namely, by exploiting the access-delegation framework OAuth 2.0 to control the accesses to a microservice.

Publicly Accessible Microservices. This smell occurs when a microservice can be invoked directly by an external client. This would require authentication to be performed on all publicly accessible microservices, always requiring users' credentials, which may compromise the application's *Confidentiality* (e.g., by exposing long-term credentials). It would also result in increasing an application's overall attack surface and in reducing its usability and maintainability. For this reason, the suggested refactoring is to *Add an API Gateway* to be used as the entry point of the application. Individual microservice APIs can then be accessed through the gateway, and authentication can be performed centrally, overall reducing the attack surface of the application and simplifying authentication-auditing tasks.

Unnecessary Privileges to Microservices. This smell occurs when unnecessary privileges are granted to a microservice, even if it does not need them to deliver its business functions. If this is the case, the resources of an application may become unnecessarily exposed, therefore increasing the attack surface and likelihood of compromising the application's *Confidentiality* and *Integrity*. A microservice application should rather be secured by *Following the Least Privilege Principle*, namely, by ensuring that each microservice can access only the least set of functionalities and data it needs. This would indeed help contain the effects of security attacks in which attackers obtain control of a microservice, e.g., by exploiting a software vulnerability.

Own Crypto Code. Microservices should be developed by avoiding the use of *Own Crypto Code*, which may compromise an application's *Confidentiality*, *Integrity*, and *Authenticity*. The use of *Own Crypto Code* may produce a false sense of security, which may be even worse than not using any encryption solution. When the *Own Crypto Code* smell occurs, the affected microservices should be refactored by *Using Established Encryption Technologies*, namely, by replacing the own crypto code with security technologies/libraries that have already been thoroughly tested.

Non-Encrypted Data Exposure. Microservice applications may accidentally expose sensitive data, e.g., since no encryption was used when storing them or because of security vulnerabilities in the employed protection mechanisms. The *Confidentiality* and *Integrity* of the exposed data can thus be compromised, e.g., since an attacker could acquire or modify the exposed data by obtaining unauthorised access to the application. For this reason, the recommended refactoring is to *Encrypt All Sensitive Data at Rest*, with "data at rest" intended as structured/unstructured data stored in digital form (e.g., cloud storage [1]), and to decrypt such data only when it is used. Data encryption can be performed by database-management systems or by the operating system at the disk level, and it should also be applied to cached sensitive data.

Hardcoded Secrets. This smell occurs when secrets (e.g., API keys, client secrets, or long-term credentials) are hardcoded in the microservices' source code or deployment scripts. Secrets should also not be stored in environment variables, because logging platforms might accidentally expose them by dumping the environment. This could lead to *Confidentiality* and *Integrity* violations of the stored secrets. The suggested refactoring is to *Encrypt Secrets at Rest* by allowing only authorised services to access their corresponding secrets. This includes never storing credentials alongside applications, as well as avoiding passing secrets to a microservice by setting environment variables of its hosting container.

Non-Secured Service-to-Service Communications. Even if running in the same internal network, microservices should establish secure communication channels to interact with each other. When this is not the case, we have *Non-Secured Service-to-Service Communications*, which may compromise an application's *Confidentiality*, *Integrity*, and *Authenticity*. Indeed, when service communications are not secured, attackers can perform man-in-the-middle, eavesdropping, and/or tampering attacks on such communications. Development teams should rather follow a "zero-trust" approach by applying the *Use Mutual TLS* refactoring. Mutual TLS is a well-known method to secure service-to-service communications that creates a secure communication channel that features data encryption and mutual authentication, hence preventing, e.g., man-in-the-middle attacks.

Unauthenticated Traffic. This smell occurs when a microservice receives unauthenticated requests from external clients or other microservices of the same application. This may result in violating the application's *Authenticity*, since there is no guarantee that the exchanged data are trusted and have not been modified. The suggested refactoring is to *Use Mutual TLS* (described earlier) and to *Use OpenID Connect*, an identity layer built on top of OAuth 2.0.

Multiple User Authentication This smell occurs on microservice applications authenticating users through multiple different access points. Each access point can be exploited by intruders to authenticate as end-users, thus violating the *Authenticity* of the application. The suggested refactoring is to *Use Single Sign-On*, namely, a single entry point to handle user authentication. The single sign-on can be realised by also implementing the *Add an API Gateway* and *Use OpenID Connect* refactorings discussed earlier.

Centralised Authorisation This smell occurs when authorisation is handled by only one application component. This may result in violating the *Authenticity* of the application, since compromising such a component could lead to an attacker having full access to the application. A fine-grained, microservice-level authorisation control should rather be enforced. For this reason, the suggested refactoring is to *Use Decentralised Authorisation* by relying on access tokens (e.g., JSON Web Tokens) transmitted with each request. Tokens indeed provide a mechanism to pass digitally signed user claims or data, which enables the enactment of access control at the microservice level.

3. Related Work

Securing microservice applications is crucial [5], as also witnessed by the number of recent contributions in the field. For instance, Chondamrongkul et al. [9] propose a technique to automatically identify security threats in microservice applications, starting from a collection of formally defined security characteristics. Other examples are given by Schneider et al. [10] and Zdun et al. [11], who support checking microservice applications for security by extracting security-aware dataflow diagrams and providing KPI metrics to measure the effectiveness in securing applications, respectively. Automatically detecting the security smells proposed by Ponce et al. [1] in microservice applications is, however, still an open problem [8], as also witnessed by the recent review by Pinheiro et al. [12]. Our objective is therefore to complement the existing studies on microservices' security smells [1,8] by providing concrete techniques for their detection and a first tool for support.

At the same time, there exist methods and tools for analysing microservice applications' security that can also be used to detect other security smells, sometimes partly overlapping with those of Ponce et al. [1]. The closest to our proposal is perhaps the security smell detection proposed by Rahman et al. [13]. They propose a static analysis approach to detect smells in Infrastructure-as-Code (IaC) scripts. Their approach differs from ours in its objectives, as it aims at detecting IaC security smells, rather than microservices' security smells. The security smells detected by Rahman et al. [13], however, cover two instances of microservice security smells that we also consider, viz., *Hardcoded Secrets* and *Unnecessary Privileges to Microservices*, which can be detected on the IaC scripts used to deploy a microservice application. The other microservice security smells by Ponce et al. [1] cannot be detected with the method proposed by Rahman et al. [13].

There also exist production-ready tools for security analysis, such as Kubesec.io [14], Checkov [15], Kube-bench [16], Kube-hunter [17], OWASP Zed Application Proxy (ZAP) [18], OpenAPI-fuzzer [19], and Sonarqube [20], for instance. Kubesec.io [14] implements a static analysis of Kubernetes configuration files, aimed at detecting known security weaknesses. More generally, Kubesec.io [14] provides a priming tool to detect misconfigurations that may induce vulnerabilities for a microservice application deployed with Kubernetes.

Checkov [15] is another tool based on the static analysis of deployment configuration files. It enables the detection of security weaknesses due to misconfigurations in the Kubernetes configuration files and Dockerfiles, among others. Checkov features many built-in policies, each targeting a specific known security weakness, and it provides a Python interface to support extending it with custom policies.

Kube-bench [16] and Kube-hunter [17] instead implement two different dynamic analyses to detect security issues in containerised applications running on some Kubernetes clusters. Kube-bench [16] checks whether a running application adheres to the security guidelines defined by the Center for Internet Security [21]. This is carried out by inspecting the configuration of the cluster where the application is running by running Kube-bench either directly on its nodes or as a privileged pod on the cluster. Kube-hunter [17] instead performs a black-box vulnerability assessment of the cluster. Kube-hunter can also run a so-called "active scan" that will try to automatically exploit the vulnerabilities found to find even more vulnerabilities to automatically create exploitation chains.

OWASP ZAP [18] and OpenAPI-fuzzer [19] are two other tools enacting dynamic analyses for finding the security vulnerabilities/weaknesses of running applications. They, however, differ from the above-described tools in their scope and method, focusing on the APIs of running services and enacting security testing. More precisely, OWASP ZAP [18] focuses on Web and API security testing by checking for the security risks defined by the OWASP Top Ten [22]. OpenAPI-fuzzer [19] instead exploits fuzzing to dynamically test services' APIs based on their OpenAPI specification. Even if not focusing on security only, OpenAPI-fuzzer provides a valuable supplement to security testing, thanks also to the possibility of assessing whether the implemented services' APIs conform to their corresponding specification in OpenAPI.

Finally, Sonarqube [20] is a general-purpose static code analyser aimed at enforcing code quality and security. The tool supports most used programming languages and analysis techniques, and it features static-application security testing to detect weaknesses and vulnerabilities. It defines and computes many metrics to assess the quality and the security of the source code, and so-called "quality gates" can be set to enforce a specific quality level. Sonarqube distinguishes security *hotspots*, which are code areas of high interest and security impact that may not contain any weakness, and security *vulnerabilities*, which require immediate developer action. The tool can be used to check, e.g., the OWASP Top 10-related issues [22] and it can be integrated into testing pipelines.

The above-listed security analysis tools provide validated solutions for identifying security vulnerabilities/weaknesses in existing applications and can be used for microservices applications too. They, however, differ from our proposed analyses and KubeHound in their objectives, as we instead focus on detecting the microservices' security smells of Ponce et al. [1]. At the same time, the analyses enacted by the above-listed tools can be fruitfully exploited to implement the automated detection of instances of some security smells of Ponce et al. [1], as we discuss in the following section.

4. Detecting Security Smells in Kubernetes-Based Microservices

We now demonstrate the possibility of automatically detecting the security smells for microservices listed by Ponce et al. [1]. We provide both *static* and *dynamic* analysis techniques for the purpose:

- Static analysis techniques detect security smells without running the target application, but rather by analysing the source code of an application, the Kubernetes configuration files and Dockerfiles used to deploy its microservices, or the specification of their APIs, e.g., with the OpenAPI standard [23].
- Dynamic analysis techniques instead detect security smells by inspecting and communicating with an application running on a Kubernetes cluster. The cluster may be left *unmodified*, namely, used "as-is", or it may be *modified* to enact other types of analyses, provided that the applied changes are not invalidating the analysis itself (e.g., by adding monitoring probes to analyse the traffic exchanged among microservices).

To demonstrate the possibility of automatically detecting the security smells of Ponce et al. [1], we hereafter introduce techniques enabling the detection of instances of such smells in microservice applications deployed with Kubernetes, the de facto standard for microservice orchestration [5]. The presented techniques are also designed to enable the reuse of existing security analysis tools (such as those recapped in Section 3) when possible.

For readability reasons, we separately demonstrate the detectability of each security smell. Namely, for each security smell, we introduce a set of analysis techniques named for the smell instance that they detect in Kubernetes-deployed microservice applications.

4.1. Insufficient Access Control

We consider three different instances of the *Insufficient Access Control* smell, for which we hereafter introduce three different detection techniques. The detected instances are distinguished based on whether they affect the specification of the microservice's API, their actual implementations, or their deployment in Kubernetes. In the first two cases, we take the OpenAPI standard [23] as a reference to specify microservices' API. When analysing the OpenAPI specification of microservices' API, we assume their actual implementation to align to that specified, as possible divergences can be automatically detected with existing techniques [24].

Insufficient Access Control in API Specifications. This technique enables the detection of an instance of the *Insufficient Access Control* smell by statically analysing the OpenAPI specification of microservices' APIs. This can be accomplished by identifying specific endpoints or even whole services that are not performing authorisation, according to their OpenAPI specification. The OpenAPI standard indeed allows specifying the authorisation scheme enacted by each endpoint of a microservice's API by specifying a `security` field, which links to a `securityScheme`. The latter specifies the actual authorisation scheme enforced by the endpoint, which spans from passing the username and password passed through HTTP headers, to more elaborate schemes exploiting API keys, OAuth2, or OpenId Connect [23]. The actual implementation of this analysis technique must take into consideration that the authentication endpoint itself does not by design have a security scheme associated with it. In addition, there could be some endpoints that are by design accessible without any authentication (e.g., to provide pre-authentication functionality).

Insufficient Access Control in Services' Implementations. This technique enables the detection of an instance of the *Insufficient Access Control* smell by dynamically analysing microservices' APIs, and more precisely by enacting automated API security testing. To ensure that the services perform correct access control, it is not enough to check their API specification, because the services' implementation may not fully adhere to their specification. Some form of dynamic testing is thus required to test the correctness of the authorisation implementation. This analysis does not require the modification of the cluster's structure, since it can be performed by deploying a testing pod in the cluster that calls an application's microservices and checks their response to detect possible anomalies. The testing pod can be implemented in two ways: if an OpenAPI specification is given, then OpenAPI-fuzzer [19] can be employed to check the compliance of the services' implementation with their specifications. Otherwise, OWASP ZAP can be used to detect broken access control in the services' APIs [18].

Insufficient Access Control to the Kubernetes API. This technique enables the detection of an instance of the *Insufficient Access Control* smell affecting the Kubernetes control plane by dynamically interacting with it. The Kubernetes API should indeed be configured with proper authentication and authorisation configurations, since leaving the API unprotected could lead to compromising the Kubernetes control plane, which would give an attacker full control of the deployed application. Development teams should also set up proper user/service accounts, configured with the fewest privileges to limit the risk of unauthorised access to the Kubernetes API with full privileges [25]. A tool that can be used to check whether this is the case is Kube-hunter [17], which ensures that the Kubernetes API is secured and has proper access control configured.

4.2. Publicly Accessible Microservices

We consider three different instances of the *Publicly Accessible Microservices* smell, for which we hereafter introduce three different detection techniques. The techniques are distinguished based on whether they detect smell instances by statically analysing the Kubernetes configuration files or by dynamically interacting with a running Kubernetes deployment. Extra care has to be taken in this case, as some services could be intentionally exposed by application developers, e.g., API gateways. The actual implementation of the introduced analyses should therefore enable developers to specify this information to filter out intentionally exposed services.

Exposed Kubernetes Services Using Service Type. This technique enables the detection of an instance of the *Publicly Accessible Microservices* smell by statically analysing the configuration files specifying an application's deployment in Kubernetes. In particular, for each microservice in an application, the analysis statically checks whether its deployment configuration is set to be publicly accessible. Namely, the analysis checks whether the microservice is exposed through Kubernetes `Service` objects of type `LoadBalancer` and `NodePort` [26] or by using `Ingress` objects [27].

Exposed Kubernetes Services Using External-IP Field. This technique enables the detection of an instance of the *Publicly Accessible Microservices* smell by dynamically analysing the metadata assigned by Kubernetes to running microservices. Kubernetes indeed associates all running services with a Cluster-IP, that is, an IP address to reach the service from within the internal cluster network. Publicly exposed services are also assigned an External-IP address. This analysis hence retrieves the information about all services and whether they are assigned an External-IP in their metadata.

Listing 1 gives an example of how this information can be retrieved using the command `kubectl`. The output in the figure shows the Kubernetes services running on the cluster, with two of them publicly accessible by external clients. While this may be intentional for the application's `gateway`, the same may not be the case for `service2`. The latter may indeed be an internal microservice whose misconfiguration resulted in unintentionally exposing it.

Listing 1. Example of output returned by `kubectl`, simplifed for readability reasons.

```
$ kubectl get services
NAME         TYPE         CLUSTER-IP      EXTERNAL-IP      PORT(S)
service1     ClusterIP    10.128.129.2    <none>           80/TCP
service2     NodePort     10.128.61.47    [Real IP here]   443/TCP
gateway      NodePort     10.128.61.46    [Real IP here]   443/TCP
```

Exposed Kubernetes Services Using Port Scanning. This technique enables the detection of an instance of the *Publicly Accessible Microservices* smell dynamically. This is performed by running port scanning [28], a technique that allows probing a host in order to find services listening on specific ports. The industry-standard tool that implements this technique is Nmap [29], which can also enumerate the exposed services, e.g., by returning the name of the running service. Other existing security-analysis tools, e.g., Kube-hunter [17], also perform port-scanning techniques to search for exposed services.

4.3. Unnecessary Privileges to Microservices

We consider two different instances of the *Unnecessary Privileges to Microservices* smell, for which we hereafter introduce two different detection techniques. The detected instances are distinguished based on whether they are detected by analysing the privileges assigned to the pods in Kubernetes configuration files or by checking the network policies.

Unnecessary Privileges to Kubernetes Pods. This technique enables the detection of an instance of the *Unnecessary Privileges to Microservices* by statically analysing the configuration files specifying an application's deployment in Kubernetes. The analysis checks the privileges of the cluster's pods by parsing the Kubernetes configuration files specifying them.

Kubernetes allows developers to configure each pod with some associated privileges, e.g., through the `securityContext` field, which describes the security properties that a deployed pod should have (Listing 2). This field can be given for each pod or each container in a pod, enabling a more granular privilege specification. The properties specified include, e.g., if the service should run as privileged or unprivileged, its access control to files using User ID and Group ID, and its Linux capabilities. A proper setting of the `securityContext` field can thus help achieve minimal privileges for a microservice at the level of the Kubernetes pod used to deploy it [30]. Implementations of this analysis should ensure that the service is running as a non-privileged user, the application has minimal capabilities, and the host network namespace is invisible to the service.

Listing 2. Example configuration of `securityContext`.

```
securityContext:
  runAsNonRoot: true
  runAsUser: 10001
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  capabilities: { drop: [ all ], add: [ NET_BIND_SERVICE ] }
```

Moreover, the proper configuration of service accounts should be taken into careful consideration. If this is not carried out, an attacker that obtained unauthorised access to a pod's file system, e.g., by exploiting a software vulnerability, may also be able to escalate their privileges by gaining access to the Kubernetes API. Service accounts offer a solution to try to mitigate the impact of this occurrence by offering the pods a token to access the Kubernetes API, if needed, and with limited privileges [31]. The described analysis is implemented and validated in two existing tools, viz., Kubesec.io [14] and Checkov [15].

Misconfigured Network Policies. This technique enables the detection of an instance of the *Unnecessary Privileges to Microservices* smell by dynamically checking whether each service can communicate only with the services it is supposed to. In Kubernetes, developers can configure network policies. These are often used in combination with Kubernetes namespaces to create groups of microservices that can communicate with each other, but not with other entities [32]. This enables creating pods with the fewest privileges possible, as they can access only the services/resources they require to provide their functionality. The tool Kube-bench [16] can be used to dynamically check such network policies.

4.4. Own Crypto Code

We consider two different instances of the *Own Crypto Code* smell, for which we hereafter propose two different detection techniques. The detection techniques are distinguished based on whether they search for the occurrence of cryptographic primitives (https://csrc.nist.gov/glossary/term/cryptographic_primitive, accessed on 26 June 2023) or suspicious cryptographic names.

Usage of Cryptographic Primitives. This technique enables the detection an instance of the *Own Crypto Code* smell by statically analysing the microservice's source code to detect the use of cryptographic primitives, e.g., AES or RSA. The usage of such primitives could be a symptom of a custom cryptography solution being re-implemented, instead of reusing established and trusted high-level solutions. One such detection of primitive cryptographic function calls can be enacted using the tool Sonarqube [20], which tags as a "hotspot" any invocation of cryptographic primitives. This is not to be confused with the usage of weak and outdated cryptographic systems (e.g., the DES cryptosystem), which is considered by Sonarqube a security vulnerability.

Suspicious Cryptographic Names. This technique enables the detection of an instance of the *Own Crypto Code* smell by statically analysing the microservice's source code to detect suspicious cryptographic names used therein, e.g., `RSA`, `IV`, or `AES`, which could suggest some customised cryptographic implementation. This of course does not mean that the code performs custom cryptography computation, but it at least informs the developers that the naming of the function could be misleading.

4.5. Non-Encrypted Data Exposure

We consider a possible instance of the *Non-Encrypted Data Exposure* smell, which occurs when data-at-rest encryption is not enabled in data stores. We therefore introduce the following analysis to detect one such smell instance.

Data-at-Rest Encryption Not Enabled in DBMSs. This technique enables the detection of an instance of the *Non-Encrypted Data Exposure* smell by statically analysing the configuration files of database management systems (DBMSs). The analysis aims at determining whether encryption-at-rest of data is enabled in a DBMS by inspecting its Kubernetes configuration files, Dockerfiles, or custom configuration files. Such a static analysis can be implemented by suitably configuring the static analysis featured by Checkov [15] to detect whether data encryption is enforced by the configuration files of the most widely used DBMSs (e.g., MongoDB, MySQL, PostgreSQL, or Redis). This analysis could be extended to work with other DBMSs not yet supported by Checkov.

4.6. Hardcoded Secrets

We consider three different instances of the *Hardcoded Secrets* smell, for which we hereafter introduce three different detection techniques. The introduced techniques are distinguished based on whether the secrets are hardcoded in Kubernetes configuration files, Dockerfiles and source code, or in containers' environment variables.

Hardcoded Unencrypted Kubernetes Secrets. This technique enables the detection of an instance of the *Hardcoded Secrets* smell by statically analysing the configuration files specifying an application's deployment in Kubernetes and by looking for unencrypted secrets written therein. As stated in the official documentation, Kubernetes secrets are by default stored unencrypted, typically represented in base64 encoding, which can be decoded to recover the original secret [33]. Listing 3 shows an example of an unencrypted Kubernetes secret in which one can readily recover the username "`admin`" and password "`password`" by decoding the `data` fields. The tool Checkov [15] can statically detect unencrypted Kubernetes secrets in order to enforce encryption-at-rest of secrets.

Listing 3. Example of an unencrypted Kubernetes secret.

```
apiVersion: v1
data: { username: YWRtaW4=, password: cGFzc3dvcmQ= }
kind: Secret
type: Opaque
metadata: { ... }
```

Hardcoded Secrets in Dockerfiles and Source Code. This technique enables the detection of an instance of the *Hardcoded Secrets* smell by statically looking for hardcoded secrets in the Dockerfiles and source code of the microservices forming an application. Since it is not obvious whether secrets are stored, we can employ different techniques to find them. For instance, we can define heuristics on the names of constants (e.g., "user", "password", "pwd", etc.), match the file's content against regular expressions that represent known secrets formats (e.g., AWS API keys format), or use entropy-based approaches. This analysis technique is very common among existing tools' implementations, and it is performed in both Checkov [15] and Sonarqube [20].

Hardcoded Secrets in Containers' Environment. This technique enables the detection of an instance of the *Hardcoded Secrets* smell by dynamically inspecting the containers running an application's microservices. The analysis looks for hardcoded secrets within the environment variables of such containers. The environment variables can be retrieved from the containers running in the cluster's pods by exploiting the Kubernetes API with the command

```
$ kubectl exec [pod_name] --env.
```

Alternative approaches may exploit the `docker inspect` command, which also allows the inspection of the environment variables set for a running container. The retrieved environment variables are then checked to detect *Hardcoded Secrets* using the same heuristics as those described for the *Hardcoded Secrets in Dockerfiles and Source Code* analysis.

4.7. Non-Secured Service-to-Service Communications

We consider one possible instance of the *Non-Secured Service-to-Service Communications* smell, which can be detected with the analysis introduced hereafter.

Unencrypted Pod-to-Pod Traffic. This technique enables the detection of an instance of the *Non-Secured Service-to-Service Communications* smell by dynamically analysing the traffic exchanged among the pods used to deploy an application's microservices. The objective of the analysis is indeed to detect unencrypted traffic that is exchanged among such pods. This analysis requires the modification of the cluster by deploying network probes to record the traffic exchanged among the microservices forming the deployed application. The captured packets are then analysed to ensure that all the traffic is encrypted, reporting the cases in which packets are exchanged by exploiting unencrypted communication protocols, such as HTTP or HTTP2, for instance.

4.8. Unauthenticated Traffic

We consider one possible instance of the *Unauthenticated Traffic* smell, which can be detected by looking for user context drops in internal requests.

User Context Drop in Internal Requests. This technique enables the detection of an instance of the *Unauthenticated Traffic* smell by dynamically analysing the distributed traces of microservice interactions to detect a drop in user context in such traces. In microservice applications, there is the need to share the calling user context among the microservices, which is used to propagate user claims in a decentralised manner and to ensure the authenticity of requests. Typically, this is achieved by propagating a signed token (often a JSON Web Token) to each involved service. Such a service can then verify the signature and enact access control based on the user claims contained in the token.

The analysis makes individual requests to the target application and, by exploiting network probes in the cluster, traces the call graph of the request. The user context drop occurs if, at some point in the call graph, requests do not have the user context token anymore. This is an instance of the *Unauthenticated Traffic* smell, since (starting from the service that dropped the user context) requests are no longer authenticated.

4.9. Multiple User Authentication

We consider three different instances of the *Multiple User Authentication* smell, for which we hereafter introduce three different detection techniques. The techniques enable the detection of such smell instances in the OpenAPI specification of microservices' APIs or by interacting with token-emission endpoints.

Multiple Authentication Endpoints in Services' API specifications. This technique enables the detection of an instance of the *Multiple User Authentication* smell by statically analysing the specification of the microservices' APIs in the OpenAPI standard [23]. The analysis detects multiple endpoints that enact user authentication, as per the specifications in the `security` field in the OpenAPI specification (described in Section 4.1). If the analysis detects multiple endpoints secured through HTTP basic authorisation (i.e., username and password), it considers this as an instance of the *Multiple User Authentication* smell.

Suspicious Authentication Endpoints and Parameters. This technique also enables the detection of an instance of the *Multiple User Authentication* smell by statically analysing the specification of the microservices' APIs in the OpenAPI standard [23]. The analysis, however, focuses on identifying, from the OpenAPI specification of the microservices, endpoints with suspicious names and parameters that would suggest that such endpoints are performing authentication. These endpoint names can be things such as `/auth` or

`/login`, and parameter names can be things such as `username` or `password`. The *Multiple User Authentication* smell is detected if there are multiple such endpoints in an application.

Multiple Access Token Emission Endpoints. This technique enables the detection of an instance of the *Multiple User Authentication* smell by dynamically looking for multiple endpoints that emit an access token back to the user. The emission of a token is common practice for standard authentication methods such as OAuth2 and OpenID Connect. Following the single-sign-on pattern, there should be only one endpoint that can emit such tokens. Therefore, this dynamic analysis can make requests to the application and analyse the responses to detect if they contain an access token. If we detect multiple endpoints emitting an access token, we have an instance of the *Multiple User Authentication* smell.

4.10. Centralised Authorisation

Instances of the *Centralised Authorisation* smell can be detected with the *User Context Drop in internal requests* analysis described in Section 4.8. Indeed, if authorisation is performed only at the edge of the application, the user context will not be propagated through the internal services, which can be interpreted as a user context drop by the gateway. Instances of the *Centralised Authorisation* smell can also be detected by using the *Insufficient access control in Services' APIs* analysis described in Section 4.1, detecting missing authorisation in the internal services' APIs. We here identify another possible instance of the *Centralised Authorisation* smell, for which we provide the following detection technique.

Central Request Point. This technique enables the detection of an instance of the *Centralised Authorisation* smell by dynamically checking whether there exists a single microservice that is invoked for each request made to an application. *Centralised Authorisation* indeed also occurs if the application is using a single microservice to perform authorisation. The analysis could make requests to the application and then, for each one of them, we inspect the call graph of the internal services. If there is one service that is always called (other than the edge service, which will likely be the API gateway) we can suspect that it is performing authorisation. As with the *User Context Drop in internal requests* analysis (Section 4.8), this analysis requires modification of the cluster by inserting network probes and enabling access to the network packet content.

4.11. Summary

Table 1 recaps the techniques introduced in the previous sections to enable automatic detection of instances of the microservice security smells of Ponce et al. [1].

The table associates each analysis technique with its type and its needed inputs. It hence distinguishes the static analysis techniques processing the application's source code, deployment files, or OpenAPI specification from the dynamic analysis techniques interacting with a modified/unmodified cluster where the application is running. Table 1 also indicates the security smells that can be detected with each analysis technique and the existing tools that can be used to implement such techniques, if any.

The analyses listed in Table 1 provide examples of techniques enabling the detection of different instances of the microservices' security smells of Ponce et al. [1] to demonstrate that their detection can be automated. At the same time, being examples, such analysis techniques are not intended to comprehensively cover all the possible instances of such smells. For instance, we consider the OpenAPI standard for microservices' API specifications, if available, but APIs might be specified in different formats and therefore require different analysis techniques. For future work, we plan to expand the set of analysis techniques enabling the automatic detection of instances of the microservices' security smells, with the ultimate goal of enabling a more comprehensive and effective detection of microservice security smells. This is one of the main reasons behind developing an extensible tool for microservice smell detection, like the one we introduce in the following section.

Table 1. Analyses to automatically detect microservices' security smells.

Name	Type	Input	Detected Smells	Tools
Insufficient Access Control in API Specifications	static	OpenAPI	*Insufficient Access Control, Centralised Authorisation*	-
Insufficient Access Control in Services' Implementations	dynamic on unmodified cluster	OpenAPI	*Insufficient Access Control*	OpenAPI-fuzzer
Insufficient Access Control to the Kubernetes API	dynamic on unmodified cluster	-	*Insufficient Access Control*	Kube-hunter
Exposed Kubernetes Services using service type	static	Kubernetes configuration files	*Publicly Accessible Microservices*	-
Exposed Kubernetes Services using External-IP Field	dynamic on unmodified cluster	-	*Publicly Accessible Microservices*	-
Exposed Kubernetes Services using Port Scanning	dynamic on unmodified cluster	-	*Publicly Accessible Microservices*	Nmap, Kube-hunter
Unnecessary Privileges to Kubernetes Pods	static	Kubernetes configuration files	*Unnecessary Privileges to Microservices*	Kubesec.io, Checkov
Misconfigured Network Policies	dynamic on unmodified cluster	-	*Unnecessary Privileges to Microservices*	Kube-bench
Usage of Cryptographic Primitives	static	source code	*Own Crypto Code*	Sonarqube
Suspicious Cryptographic Names	static	source code	*Own Crypto Code*	-
Data-at-Rest Encryption Not Enabled in DBMSs	static	Kubernetes configuration files, Dockerfile	*Non-Encrypted Data Exposure*	Checkov
Hardcoded Unencrypted Kubernetes Secrets	static	Kubernetes configuration files	*Harcoded Secrets*	Checkov
Hardcoded Secrets in Dockerfile and Source Code	static	Dockerfile, source code	*Harcoded Secrets*	Checkov, Sonarqube
Hardcoded Secrets in Containers' Environment	dynamic on unmodified cluster	-	*Harcoded Secrets*	-
Unencrypted Pod-to-Pod Traffic	dynamic on modified cluster	-	*Non-Secured Service-to-Service Communications*	-
User Context Drop in internal requests	dynamic on modified cluster	-	*Unauthenticated Traffic, Centralised Authorisation*	-
Multiple Authentication Endpoints in Services' API specifications	static	OpenAPI	*Multiple User Authentication*	-
Suspicious Authentication Endpoints and Parameters	static	OpenAPI	*Multiple User Authentication*	-
Multiple Access Token Emission Endpoints	dynamic on unmodified cluster	-	*Multiple User Authentication*	-
Central Request Point	dynamic on modified cluster	-	*Centralised Authorisation*	-

5. An Extensible Tool for Detecting Microservices' Security Smells

We hereafter introduce KubeHound, a prototype tool for automatically detecting security smells in microservice applications. More precisely, after describing the extensible architecture of KubeHound (Section 5.1), we present its prototypical implementation (Section 5.2) and that of the six smell-detection techniques it already features (Section 5.3).

5.1. Architecture of KubeHound

The main goal of KUBEHOUND is to enable automation of the detection of the microservices' security smells of Ponce et al. [1] by running analysis techniques such as those outlined in Section 4. KUBEHOUND therefore takes as input the Kubernetes configuration files and Dockerfiles specifying the target application deployment and a configuration file specifying where to retrieve the microservices' source code and their API specifications in the OpenAPI standard, when available. The Kubernetes cluster to which it connects to run dynamic smell-detection techniques is instead automatically retrieved from the `KUBECONFIG` variable specified in the environment where KUBEHOUND is running.

KUBEHOUND then processes the above-listed inputs by running the featured smell-detection techniques. In this perspective, with the examples of possibly analysis techniques outlined in Section 4, a main requirement of KUBEHOUND is *extensibility*, to enable a more comprehensive coverage of the possible instances of already-known security smells [1], as well as to enable the incorporation of newly discovered security smells, if any. Newly implemented analysis techniques must be easily pluggable into KUBEHOUND, and it must be possible to implement them by reusing or integrating with existing security-analysis tools (as illustrated for some of the analysis techniques in Section 4).

To realise the required extensibility, we designed KUBEHOUND with the *Plugin Architecture* [34] in Figure 2. KUBEHOUND is composed of three core components, viz., `Hound`, `Frontend`, and `Scheduler`, and a set of analysis plugins, viz., $\mathtt{A}_1, \mathtt{A}_3, \ldots \mathtt{A}_n$. Each analysis plugin is responsible for implementing one or more techniques for detecting instances of microservices' security smells. In general, analyses should be specific to a given instance of a security smell, as the overall detection capabilities of KUBEHOUND are derived from combining the results of all featured analyses.

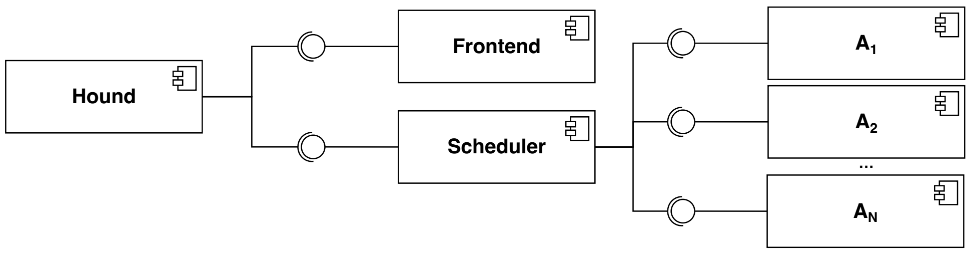

Figure 2. Architecture of KUBEHOUND, modelled as a UML component diagram [35].

The core components are instead responsible for acquiring the configuration files specifying an application deployment, parsing them, scheduling the analysis plugins implementing the featured smell-detection techniques, and collecting their results. More precisely, `Hound` takes the input files and orchestrates the smell detection on the microservice application specified therein. It first passes the input files to the `Frontend`, which is responsible for application data acquisition and parsing. `Frontend` then returns to `Hound` a collection of application objects, each representing a resource in the application's sources, together with some metadata about them. The application objects and metadata are then passed to the `Scheduler`, which is responsible for invoking the plugins to run the analyses they implement, as well as for collecting and suitably merging their results. The merged results are then returned to `Hound`, which displays them to the end-user.

Each analysis plugin complies with a given interface, which includes the properties it should expose, e.g., the name, the description, and the required application objects. The `Scheduler` exploits such properties to prepare the input to each analysis and call them accordingly using the plugin interface. KUBEHOUND's `Scheduler` follows a "best effort" approach, meaning that it invokes a plugin only if the required inputs were provided by the end-user. If this is not the case, the plugin's execution is skipped, as the analysis technique it implements cannot be performed. As a result, KUBEHOUND always performs the largest possible subset of smell-detection techniques it can, given the inputs provided by the end-user. This has two advantages. On the one hand, it enables the development of plugins without making assumptions about the structure and/or technologies of the microservice application they will analyse. On the other hand, it enables end-users to provide only what they wish to be analysed, e.g., enabling KubeHound to work if the source code of some microservices is not available or if the users do not wish KubeHound to interact with the Kubernetes cluster where an application is running.

5.2. Implementation of KubeHound

KubeHound is open-source and publicly available on GitHub (https://github.com/di-unipi-socc/kube-hound, accessed on 26 June 2023). A snapshot of the current version of the tool is also stored on Software Heritage at https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/di-unipi-socc/kube-hound, accessed on 26 June 2023). It is implemented in Python, and this is mainly motivated by the fact that Python already features a rich ecosystem of libraries, including those to process/interact with Docker and Kubernetes [36].

Input Configuration File. KubeHound takes as input a YAML configuration file structured in four main YAML objects, viz., `repositories`, `deployment`, `services`, and `properties`, as shown in Listing 4.

- The object `repositories` provides a list of named repositories storing the deployment configuration files and source code of the target microservice application. The listed repositories can be local folders or remote git repositories, which will be cloned locally by KubeHound when starting to analyse the target application. The need to organise the application source in multiple logical repositories comes from the fact that many microservice applications are developed in a "one repository per microservice" fashion, so tracking the different sources for different microservices' files requires an additional layer of abstraction.
- The object `deployment` indicates the repository where the configuration files specifying the target application deployment is found by also supporting regular expressions to match all deployment configuration files. Currently, only `kubernetes` configuration files are supported, but the syntax is intended to allow for extending the support to other deployment frameworks.
- The object `services` lists all the microservices forming the target application by associating them with the paths to their related configuration files, source code, and the specification of their APIs in OpenAPI, if available.
- The object `properties` provides a list of claimed properties for the services, which can be used by KubeHound to ignore some detected smells to reduce the amount of false positive reports. Currently, two properties are supported, viz., `external`, to indicate that a service is intentionally exposed to external clients, and `performsAuthentication`, to indicate that a service is known to perform authentication.

KubeHound will ignore any missing field in the configuration file, and it will apply a "best effort" approach to provide meaningful results out of the provided resources.

Application Acquisition and Parsing. The repositories described in the configuration files are first acquired and parsed by KubeHound. For local folders, the acquisition phase is not present, while remote git repositories are cloned locally in the current working directory. If a folder with the same name already exists, KubeHound will try to detect if it is a git repository and will perform a `pull` command to fetch the latest version of the repository. This provides a mechanism to avoid always downloading the full application on each run of the tool, which can be costly for large code bases.

Once all the files are gathered, KubeHound will perform the parsing of the application, which means taking the application files and turning them into a collection of application objects. An application object is an object that represents any resource in the application. They have three main fields: `type`, `path`, and `data`. `type` specifies the type of object (viz., `kubernetes_config`, `dockerfile`, or `openapi`), `path` specifies the path of the corresponding application file, while `data` specifies optional metadata of the object.

Listing 4. Example of input configuration file for KUBEHOUND, with some repositories/services omitted for readability reasons.

```
repositories:
  main: { git: https://github.com/microservices-demo/microservices-demo.git }
  front-end: { git: https://github.com/microservices-demo/front-end.git }
  shipping: { git: https://github.com/microservices-demo/shipping.git }
  ...

deployment:
  kubernetes: { repository: main, glob: deploy/kubernetes/manifests/*.yaml }

services:
  - { name: front-end, image: weaveworksdemos/front-end:0.3.12, repository: front-end,
     dockerfile: Dockerfile }
  - { name: shipping, image: weaveworksdemos/shipping:0.4.8, repository: shipping, dockerfile:
     docker/shipping/Dockerfile }
  ...

properties:
  front-end: { external: true }
```

For Dockerfiles and OpenAPI specifications, application objects map one-to-one to a particular file. For Kubernetes configuration files, instead, multiple application objects can be generated from the same file, one for each YAML document. KUBEHOUND will also put in the `data` object the claimed properties specified in the configuration file. KUBEHOUND will also follow a "best effort" approach for application parsing, so if there are missing resources or no metadata available for a given object, it will simply be ignored.

Analysis Scheduling and Results. The `Scheduler` component prepares the inputs for all the available analyses, runs them, and collects their results. For this to happen, analysis techniques must be implemented as Python classes extending the existing `StaticAnalysis` or `DynamicAnalysis` classes, as exemplified in Listing 5. The analyses expose some parameters through class variables, such as their name or their description, and implement their functionality through the `run_analysis` method. They also expose the `input_types` variable, which specifies the types of input this analysis requires. Any particular analysis is called only if there is at least one available object of the required types, passing them as a list of application objects to the `run_analysis` method. Custom analyses can be added to the scheduler using the `register_analysis` method (as shown in Listing 5).

Each analysis is expected to complete in a finite time, and this is particularly important for dynamic analyses needing to run for longer periods, e.g., to first monitor and then analyse the traffic exchanged among deployed microservices (as described in Section 4.7). Upon completion, the analysis returns a list of results, represented as Python objects. The results returned by all the run analysis plugins are collected together by the `Scheduler`, and they are suitably merged to compute the union of all detected smell instances. KUBEHOUND then returns the merged results as the output of the overall analysis by streamlining them to a JSON representation.

5.3. Implementation of Selected Analyses

We hereby illustrate the implementation of six of the analysis techniques described in Section 4, which come as built-in plugins in KUBEHOUND. The selection of which analysis plugins to present was mainly aimed at showcasing the different types of techniques, while also demonstrating the possibility of implementing them by integrating them with existing analysis tools. For static analysis techniques, we present the plugins implementing the detection of *Insufficient Access Control in API Specifications* and *Multiple Authentication Endpoints in Services' API Specifications*, both analyzing the OpenAPI specifications. We also present the detection of *Unnecessary Privileges to Kubernetes Pods*, which we realised by integrating the tool Kubesec.io [14]. For dynamic analysis techniques, instead, we present the detection of *Hardcoded Secrets in Containers' Environment* and the detection of *Exposed Kubernetes Services using External-IP Field*, both of which are custom dynamic techniques applied to an unmodified cluster. We also present the plugin implementing the detection of *Unencrypted Pod-to-Pod Traffic*, which exploits network probes, hence being a dynamic technique applied to a modified cluster.

Listing 5. Example of Python script running KubeHound with an additional custom analysis (lines 14–27). The custom analysis is a static analysis called `HelloWorldAnalysis` (line 1), and it inputs the Kubernetes configuration files and Dockerfiles of the target application (line 6). When invoked, it prints the string "Hello, World!" and returns an empty list of detected smells (lines 8–12).

```
 1 # create a new static analysis that prints Hello, World!
 2 class HelloWorldAnalysis(StaticAnalysis):
 3     analysis_id = "hello_world"
 4     analysis_name = "Hello World Analysis"
 5     analysis_description = "This analysis prints Hello, World!"
 6     input_types = ["kubernetes_config", "dockerfile"]
 7 
 8     def run_analysis(self, input_objects: Mapping[str, List[ApplicationObject]])\
 9             -> List[AnalysisResult]:
10         # actual analysis code goes here
11         print('Hello, World!')
12         return []
13 
14 # instantiate the Hound object and set the configuration path
15 hound = Hound(Path("test_files/mock-application/"))
16 hound.set_config_path(Path("test_files/mock-application/mock-config.yaml"))
17 
18 # acquire and parse the application
19 hound.aquire_application()
20 hound.parse_application()
21 
22 # register HelloWorldAnalysis to the scheduler
23 hound.register_analysis(HelloWorldAnalysis)
24 
25 # run the analyses and show the results
26 hound.run_analyses()
27 hound.show_results()
```

Insufficient Access Control in API Specifications. We implemented the *Insufficient Access Control in API Specifications Analysis* (Section 4.1) as a plugin to KubeHoundto enable it to detect insufficient access control using the services' OpenAPI specifications (https://github.com/di-unipi-socc/kube-hound/blob/master/kube_hound/builtin_analyses/insufficient_access_control_openapi.py, accessed on 26 June 2023). The plugin works with the 3.1.0 version of the OpenAPI specification [23], which allows it to describe security properties for endpoints, as shown in Listing 6. The plugin receives as input OpenAPI documents and starts by parsing them and searching the `securitySchemes` field, which describes all the security schemes used by the API. Then, the plugin tries to find a global `security` field, if any, and for each API endpoint, the associated `security` field. Based on the service's claimed properties, the plugin reports the security smells based on the following rules:

- If no `SecuritySchemes` is specified or, for some endpoint, no `security` field is specified, a corresponding instance of the *Insufficient Access Control* smell is reported.
- If this is the case, and if the input configuration file is not indicating that the service is `external` nor that it `performsAuthentication`, then an instance of the *Centralised Authorisation* smell is also reported.

Listing 6. Example of OpenAPI specification that describes a security scheme using HTTP bearer (i.e., `externalAPIKey`), which secures the POST method of the `/foo` path.

```
openapi: 3.0.0

paths:
  /foo: { post: ..., security: [ externalAPIKey: [] ] }
components:
  securitySchemes: { externalAPIKey: { type: http, scheme: bearer } }
```

Multiple Authentication Endpoints in Services' API Specifications. KubeHound features a plugin implementing the method described in Section 4.9, which we implemented to enable the detection of multiple user authentication endpoints based on the services' OpenAPI specifications. The implemented plugin inputs the OpenAPI specification of the APIs of the target application's microservices, and it starts by parsing them and retrieving their security properties. If multiple endpoints are recognised as using HTTP basic authentication (i.e., username- and password-based), the plugin reports an instance of the *Multiple User Authentication* smell. Services that have a corresponding `performsAuthentication` property set in the input configuration file are ignored by this plugin, since they are supposed by design to perform user authentication.

Unnecessary Privileges to Kubernetes Pods with Kubesec.io. We implemented the method described in Section 4.3 as a plugin of KubeHound to enable the detection of unnecessary privileges to Kubernetes pods (https://github.com/di-unipi-socc/kube-hound/blob/master/kube_hound/builtin_analyses/unnecessary_privileges_pods.py, accessed on 26 June 2023). To accomplish this, the implemented plugin integrates with the tool Kubesec.io [14] to perform such an analysis. Kubesec.io indeed employs a suite of test cases to check for minimum privileges in Kubernetes pod configuration files, and it comes as a Docker image on DockerHub, which we can directly pull and execute. Docker will take care of fetching (if not already present on the local machine) the Kubesec.io image, hence favouring the distribution and packaging of KubeHound. To interact with the Docker daemon, the plugin uses the Docker Python library, which exposes an API to interact with Docker containers [36]. This plugin thus starts Kubesec.io in a Docker container, which starts a web server that exposes a REST API. The plugin submits the application's Kubernetes configuration files using the `/scan` endpoint, and the Kubesec.io server then returns a JSON response containing a list of passed and failed checks. Such results are parsed and each failing test case reported by Kubesec.io is considered as an instance of the *Unnecessary Privileges to Microservices* smell.

Hardcoded Secrets in Containers' Environment. We implemented the detection of *Hardcoded Secrets in Containers' Environment* described in Section 4.6 by developing a plugin for KubeHound that exploits the Kubernetes Python client library to interact with the Kubernetes API of a cluster (https://github.com/di-unipi-socc/kube-hound/blob/master/kube_hound/builtin_analyses/hardcoded_secrets_environment.py, accessed on 26 June 2023). The plugin retrieves all the running pods of the clusters, and for each pod it retrieves all the running containers. It then executes the `env` command on each container and stores the resulting output, which is then processed to detect secrets.

For detecting secrets, the plugin uses the open-source Python module `detect-secrets` [37], which provides a built-in set of secret-detection techniques. If any environment variable is detected as storing a secret, the plugin reports the *Hardcoded Secrets* smell, together with a description of the variable that stores the secret. In the output of KubeHound the actual values are obfuscated to prevent accidental secret exposure.

Exposed Kubernetes Services using External-IP Field. We implemented the detection of *Exposed Kubernetes Services using External-IP Field* as a plugin for KubeHound that dynamically looks for exposed services by retrieving the `External-IP` field as described in Section 4.2 (https://github.com/di-unipi-socc/kube-hound/blob/master/kube_hound/builtin_analyses/exposed_services_external_ip.py, accessed on 26 June 2023). Using the Kubernetes API, the plugin retrieves the information about all the Kubernetes services running on a cluster, including its `External-IP` (if any). The service names are then matched against the user-provided properties. We report an instance of the *Publicly Accessible Microservices* smell for each service that has a non-empty external IP. This plugin makes use of the services' properties claimed in the input configuration file: if some service is declared `external`, it will assume that it was intentionally exposed to the public by the development team and it will not report any corresponding smell instance.

Unencrypted Pod-to-Pod Traffic with Ksniff. We implemented as a plugin for KubeHound to detect unencrypted pod-to-pod traffic (Section 4.7), that is, an instance of the *Non-*

Secured Service-to-Service Communications smell (https://github.com/di-unipi-socc/kube-hound/blob/master/kube_hound/builtin_analyses/unencrypted_pod_to_pod_traffic.py, accessed on 26 June 2023). In particular, the plugin focuses on detecting HTTP and HTTP2 packets, but it can be extended to work with other clear-text protocols. To achieve this, we injected network probes in the cluster to monitor traffic in the Kubernetes overlay network.

Figure 3 sketches the internals of Kubernetes networking. Each pod is assigned an IP address in the overlay network, while containers in the same pod can communicate with each other using `localhost` [38]. This implies that every container in the pod has (at least) two network interfaces: a `loopback` interface and a pod-to-pod interface, typically called `eth0` [39]. The `eth0` interface is also used for communications with external entities, e.g., health checks for the liveness of the pods [40].

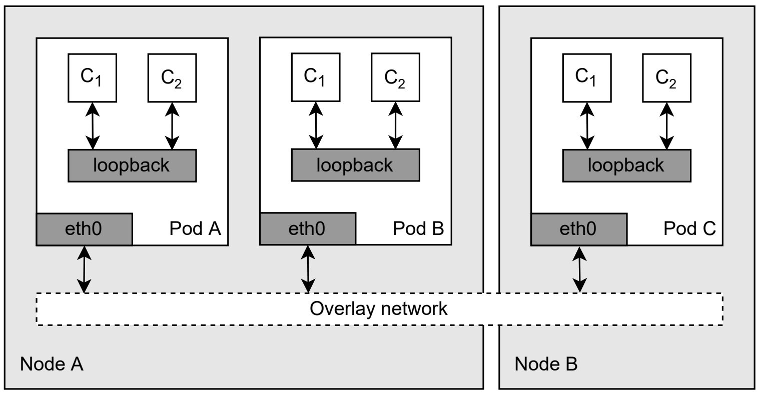

Figure 3. A sketch of the networking in Kubernetes: intra-pod communications exploit the `loopback` interface, while inter-pod communications exploit the `eth0` interface, with each pod having its unique IP in the overlay network [38].

The implementation of this plugin makes use of the tool Ksniff [41], which allows it to intercept and record traffic on clusters networked as in Figure 3. In particular, given a pod, a container, and a network interface, Ksniff will record the network traffic and it will save it on the local disk as a `pcap` file. Ksniff is run with the `-p` flag set, so as to deploy a privileged pod that has access to the node's network interfaces. This pod will run the `tcpdump` program to record network packets on the target interface.

Following the network model presented above, this plugin will capture network traffic from the `eth0` interface to intercept only pod-to-pod traffic. In the presence of a service mesh that enables mTLS via a sidecar container (e.g., the Istio service mesh), it is also possible to intercept clear-text service-to-service communications by capturing packets on the `loopback` interface [42]. This, other than providing a concrete basis to implement more sophisticated types of analysis that also take into account the content of network packets, is also an example of why services should be deployed with the fewest privileges. The takeover of a privileged pod by an attacker could result in eavesdropping of clear-text communications, even in the presence of a service mesh providing mTLS.

The plugin executes the following steps: (1) Using the Kubernetes API, the external IP addresses of worker nodes are retrieved and stored in a list. (2) For each pod related to a service in the cluster, a Ksniff privileged pod is deployed on the cluster, recording on the `eth0` interface. (3) The network traffic is recorded for a configurable amount of time (the default is 30 s) and the resulting network captures are stored in a temporary directory. (4) The Ksniff pods are destroyed to clean up the cluster. (5) Using the `pyshark` library (a Python wrapper to `tshark`), the resulting capture file is loaded and packets are filtered by excluding those whose source or destination IP is present in the nodes IP list. This is carried out to exclude them from the analysis health check requests performed by the cluster's nodes, since they are often made using plain HTTP [40]. (6) Every packet in the capture file is analysed and each packet for which `tshark` decoded the HTTP protocol layer is saved in a list. (7) If no HTTP packets are found, then the capture file is analysed again, trying to identify HTTP2 packets. The motivation for detecting HTTP2 is that high-level protocols such as gRPC are implemented on top of HTTP2 [43]. The detection of the HTTP2 protocol layer is not performed by `tshark` by default, and an additional decoding flag has to be specified during the capture loading. (8) Finally, if any HTTP or HTTP2 packets are detected, corresponding instances of the *Non-Secured Service-to-Service Communications* smell are reported, together with a sample of the packets that triggered this report.

Sometimes `tshark` will incorrectly detect the HTTP protocol layer on packets that are part of a TLS stream. In our experience, this situation occurs quite commonly, even when dealing with relatively small capture files. This could lead to false positives, as unencrypted traffic could be falsely detected. To mitigate this issue, the plugin implements a heuristic that exploits the fact that HTTP packets start with a printable line. On detection of HTTP packets, the tool will try to decode the first line as ASCII. If this decoding fails then the packet is simply ignored. In our experiments, this was sufficient to eliminate false positives without impacting the effectiveness of the smell-detection technique when unencrypted HTTP traffic was present.

6. Assessment

In this section, we report on the assessment of KubeHound by illustrating its application in controlled experiments (Section 6.1) and two case studies (Section 6.2). The controlled experiments are based on a mock application, which we realised ad hoc to enable the injection of the instances of security smells that should be detected. The two case studies are instead based on two third-party microservice applications, viz., Sock Shop [44] and Online Boutique [45]. To enable repetition of our experiments and case studies, all the necessary data are publicly available on GitHub (https://github.com/di-unipi-socc/kube-hound/tree/master/data/examples. A snapshot of the experiments' data is also stored on Software Heritage at https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/di-unipi-socc/kube-hound&path=data/examples, accessed on 26 June 2023).

6.1. Controlled Experiments

To experiment with KubeHound's functionalities, we developed a simple mock microservice application. Despite there already existing benchmarking microservice applications, such as Sock Shop [44] and Online Boutique [45], they are not focused on security aspects, but rather on experimenting with the microservice-based architectural style. Moreover, these applications are rather complex, and this makes it difficult to exploit them to "unit test" the implemented analyses. We instead decided to implement a mock microservice application that focused on security and that would adhere to well-known architectural security patterns [46,47]. The idea was to have a microservice application without any security smells and to artificially introduce them to test the analyses.

The mock application is composed of the seven microservices shown in Figure 4, each with an associated OpenAPI specification of its API. The microservices are essentially instances of the following three types of services:

- The `Authorization` server handles user tokens and authorisation.
- The `Gateway` service implements an API gateway for all the functionalities of the application.
- `Echo` services call other `Echo` services (following the topology shown in Figure 4), wait for their answers, and then simply return back the messages they received when invoked.

The functionality is therefore very simple: the gateway exposes a `/login` endpoint, where the user can perform authentication and receive an access token, and two endpoints `/echo1` and `/echo2` that simply return back the input to the user. These two endpoints must be called by the user using the authorisation token emitted by the `Authorization` server. Decentralised authorisation is implemented by using a signed user claim token emitted by the `Authorization` server, which is decoupled from the "external" authorisation mechanism [48]. A variant of this pattern was presented by Netflix in 2019, using what they called a "user passport" to solve the user-identification and -authorisation problems [49]. We also used the Istio service mesh to enable mTLS communications between the services.

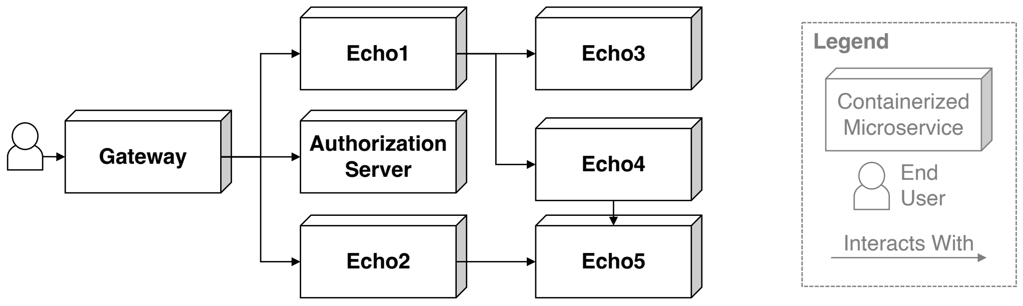

Figure 4. Microservice architecture of the mock application.

In the following, we show how we configured our mock application to inject instances of the security smells covered by the analysis techniques implemented by the plugins featured by KubeHound. Our objective is to demonstrate that all injected smell instances were automatically detected by KubeHound.

6.1.1. Insufficient Access Control in API Specifications

To test whether KubeHound can detect *Insufficient Access Control* in API specifications, we devised three alternative versions of the OpenAPI specification of the `Echo` service, which can be found online (https://github.com/di-unipi-socc/kube-hound/tree/master/data/examples/openapi_security/openapi_iac, accessed on 26 June 2023).

- The specifications in `echo_no_security.yaml` and `echo_no_schemes.yaml` instantiate two different *Insufficient Access Control* smells by not specifying the service's `security` or `securityScheme`, respectively.
- In `echo_security_override.yaml`, the `POST /echo` endpoint is overriding the global security policy with an empty policy (i.e., no authorisation is performed).

We then executed three different runs of KubeHound, to which we passed the smell-free mock application deployment. In each run, we replaced the OpenAPI specification of the `Echo` service with one of the above-listed specifications. As a result, we observed that KubeHound was capable of identifying all the injected security smells.

6.1.2. Multiple Authentication Endpoints in Services' API Specifications

For this experiment we modified the OpenAPI specification of the `Gateway` service, naming it `gateway_multiple_auth.yaml` and including an additional endpoint secured with HTTP Basic. This is an instance of the *Multiple User Authentication* smell, since the modified version has two endpoints that perform authentication. KubeHound was able to correctly identify it as an instance of the *Multiple User Authentication* smell. The configuration files used in the above-described experiment are publicly available online (https://github.com/di-unipi-socc/kube-hound/tree/master/data/examples/openapi_security/openapi_mua, accessed on 26 June 2023).

6.1.3. Unnecessary Privileges to Kubernetes Pods

To experiment with the plugin detecting *Unnecessary Privileges to Kubernetes Pods*, we generated three alternative versions of the Kubernetes deployment of `echo1`, each giving some services unnecessary privileges in a different way.

- In `echo1_no_resources.yaml`, there are no resource limits and requests specified, which could starve the cluster.
- In `echo1_no_securitycontext.yaml`, the security context of the pod is not specified.
- In `echo1_privileged.yaml`, the pod is set to be deployed in privileged mode.

All the above-listed files can be found in KubeHound's GitHub repository (https://github.com/di-unipi-socc/kube-hound/tree/master/data/examples/kubesec, accessed on 26 June 2023). We then ran KubeHound on the mock application deployment, each time considering a different alternative of the Kubernetes deployment of `echo1` among those listed above. As a result, we observed that KubeHound effectively identified all the injected smell instances.

6.1.4. Hardcoded Secrets in Containers' Environment

In this case, we added a service to our smell-free mock application deployment called `secrets-holder`. The latter is a dummy service that does not interact with the rest of the application, but just contains *Hardcoded Secrets* in its environment variables (viz., a database password, an AWS key, high-entropy hex and base64 strings, and a URL that contains basic authorisation credentials). The Kubernetes manifest file to add `secrets-holder` to the mock application deployment is publicly available online (https://github.com/di-unipi-socc/kube-hound/blob/master/data/examples/secrets_in_env/secrets-holder.yaml, accessed on 26 June 2023). We hence deployed `secrets-holder` in a Kubernetes cluster alongside our smell-free application deployment, and we ran KubeHound to analyse the services deployed on the cluster. As a result, we observed that KubeHound effectively identified all the instances of the *Hardcoded Secrets* smell that we injected into the `secrets-holder`.

6.1.5. Exposed Kubernetes Services Using External-IP Field

For this experiment, we updated the Kubernetes deployment of the `echo1` service, changing its type to a `LoadBalancer`. In this way, we prescribed it to be exposed outside of the cluster where it is deployed, hence injecting a *Publicly Accessible Microservices* smell. The updated Kubernetes deployment is publicly available on GitHub (https://github.com/di-unipi-socc/kube-hound/tree/master/data/examples/external_ip, accessed on 26 June 2023). We deployed the updated mock application on a Kubernetes cluster. We then injected another *Publicly Accessible Microservices* smell by exploiting the `kubectl expose` command to also expose the `echo2` service. Finally, we analysed the resulting application deployment with KubeHound, observing that it successfully identified both the injected instances of the *Publicly Accessible Microservices* smell.

6.1.6. Unencrypted Pod-to-Pod Traffic

This experiment aimed at checking whether KubeHound can effectively detect *Unencrypted Pod-to-Pod Traffic* occurring in the application. We deployed the smell-free mock application on the cluster, enabling and disabling mTLS between the services by using the Istio service mesh [42]. More precisely, we wrote two Kubernetes configuration files:

- `mTLS-disable.yaml`, which specifies a `PeerAuthentication` object to disable mTLS in the cluster, and
- `mTLS-strict.yaml`, which specifies `PeerAuthentication` object to enable mTLS in strict mode on the cluster.

Both the above-listed files can be downloaded from KubeHound's GitHub repository (https://github.com/di-unipi-socc/kube-hound/tree/master/data/examples/pod_to_pod_traffic, accessed on 26 June 2023). We then ran KubeHound on the cluster, making sure that we were making requests to the application while the analysis was running. When mTLS was enabled, no smells were found. Instead, when mTLS was disabled, KubeHound successfully identified the *Non-Secured Service-to-Service Communications* smells due to the unencrypted traffic being exchanged between the deployed services.

6.2. Case Studies

In this section, we describe two case studies based on two third-party microservices applications whose existing deployment was analysed by KubeHound. The applications are Weaveworks' Sock Shop [44] and Google's Online Boutique [45]. Both the considered applications are benchmarking applications intended to showcase microservice-based architectures, but not focusing on application security. Hence, we expected that some security smells could be detected therein.

6.2.1. Sock Shop

Sock Shop [44] is composed of fourteen microservices implemented in different technologies whose sources are split in multiple repositories, following the "one repository per service" pattern. The deployment scripts can be found on the main GitHub repository of the application https://github.com/microservices-demo/microservices-demo, accessed on 26 June 2023). while each microservice's source code, Dockerfile, and OpenAPI specification (if any) can be found in its own repository.

The first thing we did for this case study was create the configuration file for letting KUBEHOUND process Sock Shop's Kubernetes deployment and OpenAPI specifications. In the file, we specified the repositories from which to download the necessary inputs, and we set to `true` the `external` property of the `front-end` service, to indicate that it is acting as the gateway of the application, as indicated in the online documentation [44]. We deployed the application on a cluster, ran KUBEHOUND on the obtained application deployment, and collected its output. We only then checked whether the security smells detected by KUBEHOUND were present in the considered applications by inspecting their sources. We now comment on some of the security smells reported by KUBEHOUND on Sock Shop, while the full list of identified smells can be found online (https://github.com/di-unipi-socc/kube-hound/tree/master/data/examples/sock_shop, accessed on 26 June 2023).

Unnecessary Privileges to Microservices. By analysing Sock Shop's Kubernetes manifest files, KUBEHOUND identified various instances of the *Unnecessary Privileges to Microservices* smell in multiple services. We hence inspected such Kubernetes manifest files, observing that there are inconsistencies in the privileges assigned to the pods, which correspond to the identified smell instances. For instance, in the deployment configuration of the service `carts-db` the `securityContext` specification is incomplete, as fields such as the `runAsNonRoot` are missing and resource limits and requests are not present. Other examples are given by the deployment of the service `queue-master`, which lacks altogether any security-related specification, and by the fact that all services do not have any service account specified. All those instances of the *Unnecessary Privileges to Microservices* smells were correctly identified by KUBEHOUND.

Insufficient Access Control. The check for *Insufficient Access Control in API Specifications* enacted by KUBEHOUND identified that each microservice's OpenAPI specification did not include any `securityScheme`, hence classifying this as an instance of the *Insufficient Access Control* smell. Upon inspection of the APIs exposed by the internal services and their implementation, we realised that Sock Shop's microservices indeed do not enact any authorisation of requests, as correctly witnessed by the identified instances of the *Insufficient Access Control* smell.

Harcoded Secrets. The detection of *Hardcoded Secrets in Containers' Environment* implemented by KUBEHOUND found an interesting result, namely, that `catalogue-db`'s environment includes a variable named `MYSQL_ROOT_PASSWORD`, which stores a secret starting with `fake`. One such variable was quite suspicious, as it very likely contains the database's root password for the `catalogue-db`. We then inspected the Kubernetes deployment of the `catalogue-db` service, and we identified the hardcoded database password (viz., `fake_password`), meaning that the identified smell instance was denoting a security issue for the application.

Non-Secured Service-to-Service Communications. KUBEHOUND reported unencrypted HTTP traffic sent to or from all Sock Shop's microservices, hence identifying instances of the *Non-Secured Service-to-Service Communications* smell on all of them (Listing 7). We hence inspected the application to detect whether this truly denote some security issue, finding that this was the case, as no encryption was performed on the messages exchanged among Sock Shop's microservices.

Listing 7. Selected fragments of the output of KubeHound showing instances of the *Hardcoded Secrets* and *Non-Secured Service-to-Service Communications* smells detected on Sock Shop [44].

```
Secrets in environment variables analysis - detected smells {HS}
    Detected secret in pod catalogue-db-6d49c7c65-l46bk, container catalogue-db
    variable: MYSQL_ROOT_PASSWORD=fake*********
    reason: Secret Keyword

...

Traffic analysis - detected smells {NSC}
    Unencrypted traffic detected in pod catalogue-5f495f9cf8-wvk5f
    here is a sample of the packets (HTTP):
    HTTP 10.2.2.204 -> 10.2.2.203 : GET /catalogue HTTP/1.1
    HTTP 10.2.2.203 -> 10.2.2.204 : HTTP/1.1 200 OK
    HTTP 10.2.2.204 -> 10.2.2.203 : GET /catalogue HTTP/1.1
    HTTP 10.2.2.203 -> 10.2.2.204 : HTTP/1.1 200 OK
    HTTP 10.2.2.204 -> 10.2.2.203 : GET /catalogue/03fef6ac-1896-4ce8-bd69-b798f85c6e0b HTTP/1.1
    HTTP 10.2.2.203 -> 10.2.2.204 : HTTP/1.1 200 OK
    HTTP 10.2.2.204 -> 10.2.2.203 : GET /catalogue/d3588630-ad8e-49df-bbd7-3167f7efb246 HTTP/1.1

Traffic analysis - detected smells {NSC}
    Unencrypted traffic detected in pod front-end-6585d48b5c-qkmqp
    here is a sample of the packets (HTTP):
    HTTP 10.2.0.1 -> 10.2.2.204 : GET /catalogue HTTP/1.1
    HTTP 10.2.2.204 -> 10.128.117.95 : GET /catalogue HTTP/1.1
    HTTP 10.128.117.95 -> 10.2.2.204 : HTTP/1.1 200 OK
    HTTP 10.2.2.204 -> 10.2.0.1 : 0
    HTTP 10.2.0.1 -> 10.2.2.204 : GET / HTTP/1.1
    HTTP 10.2.2.204 -> 10.2.0.1 : rue,
    HTTP 10.2.0.1 -> 10.2.2.204 : GET /login HTTP/1.1
    HTTP 10.2.2.204 -> 10.128.133.122 : GET /login HTTP/1.1
    HTTP 10.128.133.122 -> 10.2.2.204 : HTTP/1.1 200 OK
```

6.2.2. Online Boutique

Online Boutique [45] is composed of eleven microservices, whose sources (unlike those of Sock Shop) are stored in the same GitHub repository (https://github.com/GoogleCloudPlatform/microservices-demo, accessed on 26 June 2023).

As we did for Sock Shop, we first created a configuration file for letting KubeHound process Online Boutique's Kubernetes deployment and OpenAPI specifications. In the file, we specified the repository from which to download the necessary inputs, and we set the `external` property of the `frontend` service to indicate that it is acting as the gateway of the application, as indicated in the online documentation [45]. Differently from Sock Shop, here we also had the possibility of exploiting the Istio service mesh [42] when deploying the target microservice application on a cluster. We therefore considered two different cases in our study, namely, with and without making use of the Istio service mesh, to compare KubeHound's results between the two.

We then analysed the obtained application deployments with KubeHound, whose full output is publicly available on GitHub (https://github.com/di-unipi-socc/kube-hound/tree/master/data/examples/online_boutique, accessed on 26 June 2023). The main difference between the two application deployments was in the identified *Non-Secured Service-to-Service Communications* smells: when KubeHound analysed the deployment of Online Boutique with the Istio service mesh enabled, it was not returning any instance of the *Non-Secured Service-to-Service Communications* smell due to unencrypted traffic. Instead, when analysing the deployment of Online Boutique without the Istio service mesh, instances of the *Non-Secured Service-to-Service Communications* were identified in each service interaction due to unencrypted HTTP2 traffic. We hence inspected the application, observing that the services were communicating via gRPC without encrypting the traffic themselves. The difference indeed resided in the Istio service mesh configuration, which, when enabled, enforced mutual TLS in service-to-service communications [42].

KubeHound also identified some other security smells that were common to both deployments of Online Boutique. However, compared to Sock Shop, Online Boutique was much more adherent to good security practices, so in the case of the application deployed with Istio enabled, the identified security smells did not correspond to true bad security decisions in the application. Indeed, KubeHound identified *Hardcoded Secrets* due to the featured, heuristics-based *Hardcoded Secrets in Containers' Environment* analysis, which reported false positive secrets this time. KubeHound also detected *Unnecessary Privileges to Microservices* smells due to Online Boutique's microservices running as root. However, by inspecting the Kubernetes manifest files, we observed that this was not truly the case: containers were supposed to run as non-root, but this was specified under the `spec` field instead of in the `containers` list, which is the only place where Kubesec.io looks for deployed containers' privileges.

7. Conclusions

We showed that the task of automatically detecting the security smells listed by Ponce et al. [1] is actually possible, and we presented a first set of analysis techniques enabling the detection of instances of such smells in microservice applications deployed with Kubernetes. We also provided a first tool supporting automated security smell detection, called KubeHound, which already features a subset of the presented smell-detection techniques and whose design is modular and extensible to enable extending it to provide a more comprehensive coverage of microservice security smell detection. We finally assessed our approach and tool in practice by applying KubeHound to controlled experiments and case studies based on two third-party applications.

Our work can help researchers and practitioners better understand how security smells [1] can be identified in microservice applications. They can also exploit KubeHound to automatically detect instances of such security smells in their microservice applications, if deployed with Kubernetes. The coverage of detected smell instances is, however, currently limited to the built-in plugins presented in Section 5.3. For this reason, we are already working on extending KubeHound to feature all the analysis techniques in Section 4 as built-in plugins. We also plan to investigate new analysis techniques to provide a more comprehensive coverage of the the possible instances of already-known security smells [1], as well as to enable the incorporation of newly discovered security smells, if any.

It is worth noting that KubeHound is "best effort", meaning that it runs a smell-detection technique only if all the resources it needs are available. This obviously limits the set of detected smell instances to only those that can be detected with the inputs provided to KubeHound. At the same time, it enables end-users to provide only what they wish to be analysed, e.g., enabling KubeHound to work if the source code of some microservices is not available or if users do not wish KubeHound to interact with a running instance of the target microservice application. The choice of a "best effort" approach was motivated to enhance the usability and applicability of KubeHound. We plan to further improve such usability and applicability, e.g., by devising a graphical user interface and by enabling it to work with other container-orchestration frameworks or API-specification languages. We also plan to improve the performance of KubeHound, e.g., by enabling the parallel execution of the smell-detection techniques. Finally, we plan to further support end-users by devising a confidence level to be attached to the detected security smells. This would help assess how likely is that any security smell detected by KubeHound represents a security problem in the application.

Author Contributions: Conceptualization, G.D., J.S. and A.B.; Methodology, G.D., J.S. and A.B.; Software, G.D. and J.S.; Validation, G.D.; Investigation, G.D., J.S. and A.B.; Writing—original draft preparation, J.S.; Writing—review and editing, G.D., J.S. and A.B.; Supervision, J.S. and A.B.; Project administration, A.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ponce, F.; Soldani, J.; Astudillo, H.; Brogi, A. Smells and refactorings for microservices security: A multivocal literature review. *J. Syst. Softw.* **2022**, *192*, 111393. [CrossRef]
2. Thönes, J. Microservices. *IEEE Softw.* **2015**, *32*, 116–116. [CrossRef]
3. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Softw.* **2016**, *33*, 42–52. [CrossRef]
4. Zimmermann, O. Microservices Tenets. *Comput. Sci.* **2017**, *32*, 301–310. [CrossRef]
5. Soldani, J.; Tamburri, D.A.; Van Den Heuvel, W.J. The pains and gains of microservices: A Systematic grey literature review. *J. Syst. Softw.* **2018**, *146*, 215–232. [CrossRef]
6. Yarygina, T.; Bagge, A.H. Overcoming Security Challenges in Microservice Architectures. In Proceedings of the 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE 2018), Bamberg, Germany, 26–29 March 2018; Lenhard, J., Meng, F., Wang, Y., Eds.; IEEE Computer Society: New York, NY, USA, 2018; pp. 11–20. [CrossRef]
7. Newman, S. *Building Microservices*, 1st ed.; O'Reilly: Newton, MA, USA, 2015.
8. Ponce, F.; Soldani, J.; Astudillo, H.; Brogi, A. Should Microservice Security Smells Stay or be Refactored? Towards a Trade-off Analysis. In Proceedings of the Software Architecture; Gerostathopoulos, I (ECSA 2022), Prague, Czech Republic, 19–23 September 2022; Lewis, G., Batista, T., Bureš, T., Eds.; Springer International Publishing: Cham, Switzerland, 2022; pp. 131–139. [CrossRef]
9. Chondamrongkul, N.; Sun, J.; Warren, I. Automated Security Analysis for Microservice Architecture. In Proceedings of the 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), Salvador, Brazil, 16–20 March 2020; pp. 79–82. [CrossRef]
10. Schneider, S.; Scandariato, R. Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java. *J. Syst. Softw.* **2023**, *202*, 111722. [CrossRef]
11. Zdun, U.; Queval, P.J.; Simhandl, G.; Scandariato, R.; Chakravarty, S.; Jelic, M.; Jovanovic, A. Microservice Security Metrics for Secure Communication, Identity Management, and Observability. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–34. [CrossRef]
12. Pinheiro, D.; Oliveira, J.; Figuereido, E. Microservice Smells and Automated Detection Tools: A Systematic Literature Review. In Proceedings of the 4th International Conference on Microservices, (Microservices 2022), Paris, France, 10–12 May 2022; Dorai, G., Karastoyanova, D., Osmani, A., Eds.; Microservices Community, 2022. Available online: https://www.conf-micro.services/2022/papers/paper_11.pdf (accessed on 26 June 2023).
13. Rahman, A.; Parnin, C.; Williams, L. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019), Montreal, QC, Canada, 25–31 May 2019; Bultan, T., Whittle, J., Eds.; IEEE Computer Society: New York, NY, USA, 2019, pp. 164–175. [CrossRef]
14. Kubesec.io: Security Risk Analysis for Kubernetes Resources. Available online: https://kubesec.io/ (accessed on 26 June 2023).
15. Checkov: Policy-as-Code for Everyone. Available online: https://www.checkov.io/ (accessed on 26 June 2023).
16. Kube-Bench. Available online: https://github.com/aquasecurity/kube-bench (accessed on 26 June 2023).
17. Kube-Hunter. Available online: https://github.com/aquasecurity/kube-hunter/ (accessed on 26 June 2023).
18. OWASP Zed Application Proxy. Available online: https://www.zaproxy.org/ (accessed on 26 June 2023).
19. OpenAPI Fuzzer—Black-Box Fuzzer That Fuzzes APIs Based on OpenAPI Specification. Available online: https://github.com/matusf/openapi-fuzzer (accessed on 26 June 2023).
20. SonarQube. Available online: https://www.sonarqube.org/ (accessed on 26 June 2023).
21. CIS Kubernetes Benchmark. Available online: https://www.cisecurity.org/benchmark/kubernetesCISKubernetesbenchmark (accessed on 26 June 2023).
22. OWASP. Top 10 Web Application Security Risks. 2022. Available online: https://owasp.org/www-project-top-ten/ (accessed on 26 June 2023).
23. OpenAPI Specification v3.1.0, Version 3.1.0. 2021. Available online: https://spec.openapis.org/oas/latest.html (accessed on 26 June 2023).
24. Walker, A.; Das, D.; Cerny, T. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. *Appl. Sci.* **2020**, *10*, 7800. [CrossRef]
25. Kubernetes Documentation: Authentication. Available online: https://kubernetes.io/docs/reference/access-authn-authz/authentication/ (accessed on 26 June 2023).
26. Kubernetes Documentation: Service. Available online: https://kubernetes.io/docs/concepts/services-networking/service/ (accessed on 26 June 2023).
27. Kubernetes Documentation: Ingress. Available online: https://kubernetes.io/docs/concepts/services-networking/ingress/ (accessed on 26 June 2023).
28. Bhuyan, M.H.; Bhattacharyya, D.; Kalita, J. Surveying Port Scans and Their Detection Methodologies. *Comput. J.* **2011**, *54*, 1565–1581. [CrossRef]

29. Nmap. Available online: https://nmap.org/ (accessed on 26 June 2023).
30. Kubernetes Documentation: Configure a Security Context for a Pod or Container. Available online: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/ (accessed on 26 June 2023).
31. Kubernetes Documentation: Managing Service Accounts. https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/ (accessed on 26 June 2023).
32. Kubernetes Documentation: Network Policies. Available online: https://kubernetes.io/docs/concepts/services-networking/network-policies/ (accessed on 26 June 2023).
33. Kubernetes Documentation: Secrets. Available online: https://kubernetes.io/docs/concepts/configuration/secret/ (accessed on 26 June 2023).
34. Richards, M. *Software Architecture Patterns*, 1st ed.; O'Reilly Media, Inc.: Newton, MA, USA, 2015.
35. OMG. Unified Modeling Language (UML). 2017. Available online: https://www.omg.org/spec/UML (accessed on 26 June 2023).
36. Gift, N.; Behrman, K.; Deza, A.; Gheorghiu, G. *Python for DevOps: Learn Ruthlessly Effective Automation*, 1st ed.; O'Reilly Media: Newton, MA, USA, 2020.
37. Detect-Secrets. Available online: https://github.com/Yelp/detect-secrets (accessed on 26 June 2023).
38. Kubernetes Documentation—Services, Load Balancing, and Networking. Available online: https://kubernetes.io/docs/concepts/services-networking/ (accessed on 26 June 2023).
39. Kristijan, M. Learnk8s.io: Tracing the Path of Network Traffic in Kubernetes. Available online: https://learnk8s.io/kubernetes-network-packets (accessed on 26 June 2023).
40. Kubernetes Documentation: Configure Liveness, Readiness and Startup Probes. Available online: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/ (accessed on 26 June 2023).
41. Ksniff. Available online: https://github.com/eldadru/ksniff (accessed on 26 June 2023).
42. Calcote, L.; Butcher, Z. *Istio: Up and Running*, 1st ed.; O'Reilly Media: Newton, MA, USA, 2020.
43. gRPC over HTTP2. Available online: https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md (accessed on 26 June 2023).
44. Sock Shop—A Microservices Demo Application. Available online: https://microservices-demo.github.io/ (accessed on 26 June 2023).
45. Online Boutique. Available online: https://github.com/GoogleCloudPlatform/microservices-demo (accessed on 26 June 2023).
46. Hannousse, A.; Yahiouche, S. Securing microservices and microservice architectures: A systematic mapping study. *Comput. Sci. Rev.* **2021**, *41*, 100415. [CrossRef]
47. Washizaki, H.; Xia, T.; Kamata, N.; Fukazawa, Y.; Kanuka, H.; Kato, T.; Yoshino, M.; Okubo, T.; Ogata, S.; Kaiya, H.; et al. Systematic Literature Review of Security Pattern Research. *Information* **2021**, *12*, 36. [CrossRef]
48. OWASP Cheat Sheet Series: Microservice Security Cheat Sheet. Available online: https://cheatsheetseries.owasp.org/cheatsheets/Microservices_security.html (accessed on 26 June 2023).
49. User & Device Identity for Microservices @ Netflix Scale. QCon 2019. 2019. Available online: https://www.infoq.com/presentations/netflix-user-identity/ (accessed on 26 June 2023).