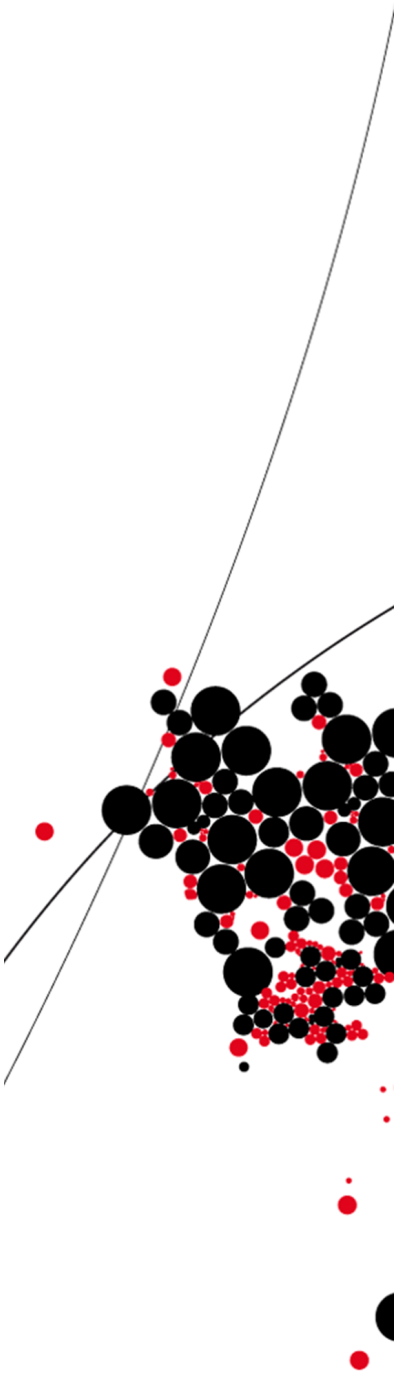
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science



HoneyKube:

Designing a Honeypot Using Microservices-Based Architecture

Chakshu Gupta

Master Thesis
August 23, 2021

Graduation Committee:
Dr.ir. Andrea Continella
Prof.Dr.ir. Roland van Rijswijk - Deij
Prof.Dr. Andreas Peter
Thijs van Ede M.Sc.

Services Cyber Security Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

HoneyKube: Designing a Honeypot Using Microservices-Based Architecture

Chakshu Gupta
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands
c.gupta@student.utwente.nl

***Abstract*—Since before the pandemic, there has been a gradual rise in the dependency on online applications. Examples include e-commerce, social media and digital conferencing. This dependence has become more pronounced during the pandemic. The user demands from these online platforms also rise with this increased dependence, which requires these applications to be agile and continuously evolve. The traditional monolithic systems do not enable this agility and make it challenging to meet these rapidly changing demands. Microservices-based architecture comes to the rescue by providing the required flexibility and agility. As we see an increase in the popularity of microservices, there is also a rise in cyberattacks targeting these environments. Because of the differences between the monolithic and microservices architectures, the traditional security solutions are not as effective in the new architecture.**

We propose a novel honeypot design with an innovative monitoring setup to facilitate data collection from cyberattacks in a system using a microservices architecture. We deploy this honeypot with a real-world application on top to attract malicious users. The data collection process with this honeypot involves two experiments: an open setting, where we expose the honeypot to the internet, and a controlled one, where we open the honeypot to a limited network. We collect approximately 850 GB of data containing the attackers' interactions with the honeypot in the two experiment settings. We evaluate the fingerprintability of the honeypot using a state-of-the-art reconnaissance tool. Moreover, we show the differences in the attack behaviour when targeting a microservices-based architecture and provide the data to further research in understanding these differences.

***Index Terms*—Microservices, Kubernetes, Honeypots, Service-Oriented Architecture**

I. Introduction

Over the past few years, we have witnessed a rise in the popularity of microservices, particularly in the development of cloud-based web applications, with tech giants like Amazon, Netflix and Uber adopting and endorsing them. Evolved from the Service-Oriented Architecture (SOA), microservices focuses on a modular architecture, breaking one enormous application into loosely coupled, independent parts. Microservices enable easier development, deployment, better maintainability, and flexible scaling of services. Software containers provide the needed mechanism for packaging a microservice with its libraries and dependencies. Containers enable the application to run reliably in any computing environment. The microservices architecture presents a significantly different setup from the traditional monolithic one that consists of single-tiered software. The various components of a monolithic architecture are interconnected and interdependent, using shared code and memory. The modular design of the microservices architecture consists of a significantly higher number of independent parts and the paths of communications connecting them. More components in a system, the larger the number of moving parts, translating to a larger attack surface. Hence securing these systems becomes much more challenging in comparison to a monolithic system.

As the popularity of this architecture increases, the attackers become more motivated to develop new and innovative methods to breach these systems. Container-based and microservices-based systems have witnessed a large number of attacks in the last few years [1]–[4].These attacks varied from infecting docker images in the Docker Hub repository [4] to developing malware that enables breaking out of the containers to establish backdoors [3]. From these attacks in containerized environments, it is evident that the approaches used to attack such systems differ from the traditional methods. Since the attack patterns differ, traditional intrusion detection systems (IDS) and firewalls developed for monolithic systems will not be as effective in identifying attacks in this new environment. Hence, it becomes vital to have security solutions tailored to identify threats in containerized and microservices settings. These solutions require data about the behaviour patterns of attackers and the tactics they employ when attempting to compromise such systems.

In 2003, the honeynet project[1] developed honeypots to facilitate data collection from genuine attacks and use it to identify attack patterns. They defined honeypots as "decoy computer resources whose value lies in being probed, attacked, or compromised" [7]. Since they are decoy systems, they have no production value and hence, any access attempt or interaction with them is considered a probe, scan or attack. The ongoing activities on these systems are logged and later analyzed to improve our understanding of the attackers' behaviour.

Academic researchers and industrial professionals have used honeypots in their respective use-cases to further their knowledge of the cyber-criminals, including their approaches and their motives [8]–[10], [14], [15], [46]–[53]. The gained insights were vital in advancing the development of defence

[1]https://www.honeynet.org/

measures. The two use-cases made use of honeypots in different contexts. In the industrial setting, honeypots are placed alongside production servers to identify the threats and vulnerabilities exploited in the wild. Whereas, in the academic setting, the purpose of the honeypots is to collect data and analyze it to identify behaviour patterns used by the attackers and the vulnerabilities exploited by them.

The amount and the quality of the data collected by the honeypot depend on the level of interaction permitted to the user in the system: low, medium, and high interaction honeypots, as well as the resemblance of the honeypot to a real system. Low-interaction honeypots consist of emulated protocols or network services without exposing the complete functionality of the operating system. They are easy to develop and deploy, but they capture a limited amount of information about the attackers' actions. On the other hand, high-interaction honeypots are real systems, imitating production systems, and designed to be vulnerable to attract attackers. Even though high interaction honeypots provide more insights into the attackers' actions, they pose a greater risk of getting used for a real cyber-attack, e.g. as part of a botnet. Hence, these can be quite expensive to deploy and maintain.

In this research, we design a honeypot named HoneyKube, using the microservices-based architecture with a real-world application on top. We deploy HoneyKube and expose it to the internet to collect attack data. The application uses Kubernetes for container orchestration and Google Kubernetes Engine (GKE) for deploying and exposing the honeypot to the internet. The monitoring system design of this honeypot consists of capturing interaction activities at multiple observation points to record data from the attacks. We aim to collect all system calls executed within the containers and all network interactions with the honeypot, including communication between the microservices.

Through this experiment, we collected nearly 850 GB of data which consists of system trace files, network trace files, and various types of log files. We recorded roughly 11500 brute-force attempts to access the system, out of which only 12 succeeded. Since all activities recorded by the honeypot tend to be malicious, the collected data can provide a good training dataset for Intrusion Detection Systems [20]–[23], such as KubAnomaly [21]. Analysis of this data will facilitate future research in understanding attackers' behaviour and attack patterns. Subsequently, it will help further the research in designing adequate defences to secure microservices-based environments.

In short, our paper makes the following contributions:

- We propose a novel honeypot design that uses microservices-based architecture. This design provides a baseline for honeypots imitating different types of real-world applications that use this architecture.
- We introduce an innovative monitoring and detection setup to record attacker movements inside a microservices-based architecture.
- We present real-world attack data collected from the honeypot. Analysis of this data will enhance our understanding of the attackers' behaviour when compromising the microservices-based architecture.

To facilitate other researchers in conducting their research in this area, we will release the source code of the honeypot on GitHub, and the collected data will be made available on request.

II. Background

In this section, we describe the microservices architecture and Kubernetes platform, a container orchestrator that automates the management of microservices.

A. Microservices

Microservices-based architecture is an application development approach, where separate components of a software design are created and deployed as isolated services. Each microservice is designed to meet a specific functional requirement, such as user management, payments, and sending emails. These microservices communicate with other services via network-based interfaces, such as remote procedural calls (RPC or gRPC for the latest high-performance framework developed by Google) and API calls. Instead of sharing a single database, like in a monolithic architecture, in this architecture, every microservice has its own database, enabling each service to use the database best suited for its requirements. This creates a loosely coupled system that allows each service to be scaled, deployed, managed, and updated independently. Software containers, with their software packaging capabilities, provide the perfect platform to host the microservices. These containers share the kernel with the host machine and use kernel features like namespaces to isolate processes while controlling resource usages such as CPU and memory. Hence, allowing each microservice to run in its custom environment, independent of other microservices.

B. Kubernetes

With the increasing complexity of applications, the number of microservices required to fulfil all their requirements also increases. This number can be in thousands, e.g., Uber's application uses more than 2000 microservices [6]. As the number increases, it becomes essential to automate the management of the microservices and the containers hosting them. Kubernetes [41], also known as K8s, is an open-source orchestrating platform that enables automating the deployment, management, and scaling of these microservices. A K8s cluster is a distributed computing setup consisting of a set of worker machines called *nodes*. Figure 1 shows the architecture of a K8s cluster and its components. A node in K8s is an abstraction for a machine, and it can be either physical or virtual. The application containers run on the nodes, as we can see in Figure 1. The *control plane* is the cluster orchestrator that handles workload distribution amongst the nodes and monitors their health. The core of the control plane is the *K8s API server* that enables the cluster administrators and other users to monitor and manage the state of the cluster. The control plane is responsible for scaling and scheduling the

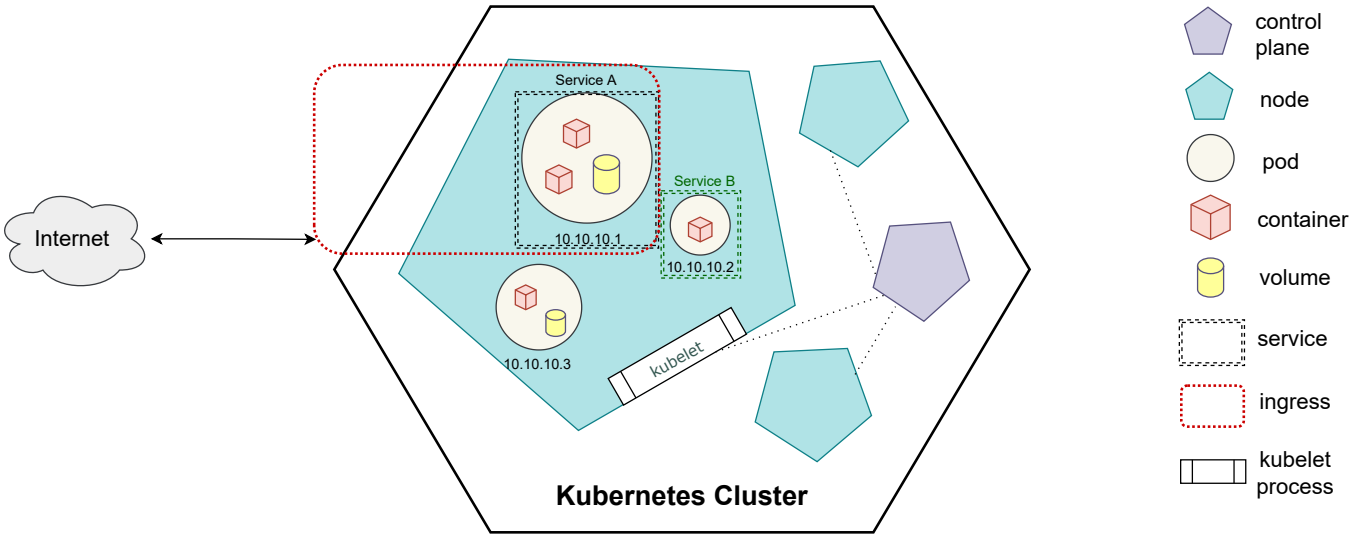


Fig. 1: **Detailed architecture of a K8s cluster. The cluster is a combination of nodes and control plane. A Node is an abstraction for a machine. The control plane is the cluster orchestrator. The dotted lines between them depict the communication channels between the nodes and the control plane. The application containers run on the nodes. A pod is an abstraction that groups containers that share network resources and volume. By default, a pod remains isolated from the outside world and requires a service resource to enable external communication. Ingress is an API object that facilitates exposing a service to the internet via a load balancer.**

application containers within the nodes and rolling out updates. The control plane communicates with the nodes using the K8s API to ensure that the containers running on them are healthy and working properly. Each node consists of an agent running on it that facilitates this communication with the control plane called *kubelet* (shown in Figure 1).

K8s employs an additional abstraction to group containers that form a microservice called a *pod*. A pod is an ephemeral resource and loses its state upon restart. Any data that requires persistence should use persistent volumes within the pods for storage. All the containers within a pod share the network resources and storage. The containers inside a pod can communicate with each other using `localhost` as though they were running on the same system while staying isolated from one another. Each pod is assigned a unique private IP address. The communication between the pods uses these IPs within the cluster. By default, the pods remain isolated from the outside world and require additional abstractions to enable external communication, such as *Service* and *Ingress*. These abstractions are depicted in Figure 1.

Because of the non-permanent nature of the pods, their IP address changes with every restart. The ever-changing IP address becomes problematic when trying to access the application from inside or outside the cluster. Service is an abstraction that helps solve this problem. Using a service, we can define a logical set of pods that work together to fulfil one design requirement and a policy containing details on how to access them. This abstraction is commonly known as a microservice. The IP address assigned to the service does not change throughout its existence. To expose these services to the outside world, we need to use *Ingress*, which is an API object that handles external access to the services within the cluster. Ingress provides load balancing, SSL termination, and name-based virtual hosting capabilities [42]. With Ingress, it is possible to expose HTTP and HTTPS routes from the internet to services inside the cluster. Figure 1 depicts the relation between Ingress and the service within the cluster.

III. Threat Model

The HoneyKube design involves a few assumptions for the extent to which the attackers can compromise the system. We use these assumptions to devise a threat model for this experiment.

- We assume the attackers aim to get access to the host machines by escaping the containers. We set up security defences to prevent this escape and any other form of privilege escalation. We describe these defences in detail in Section VI. During our evaluation, we observed that no attackers were successful in breaking out of the containers. Hence, we are more confident in this assumption.
- We assume the attackers cannot escalate privileges to gain access to other cloud resources from outside the cluster, using our Google Cloud Platform account [38], [39]. Also, we assume that they cannot shut down the experiment and use the resources for malicious purposes. We base our assumption on the default Identity and Access Management (IAM) in GCP. Every compute engine or VM in GCP uses a service account to define its

access identity, and the default service account has limited access. And, we do not give this service account any additional authority.

- We assume the attackers cannot discover or tamper with our monitoring system for HoneyKube, a common assumption in threat monitoring systems [24], [25]. For the monitoring setup, we use resources within the cluster to store the data we collect. Our assumption about tampering includes any tampering with this data. Even with these assumptions, we take some precautions (as we mention in Section V) to keep the damage to a minimum in case it happens. Hence, we assume the attackers cannot evade detection inside the system by removing the evidence of their activities [38], [39] such as clearing container logs or deleting K8s events.

The focus of HoneyKube is to capture attackers activities within the cluster on credential access, lateral movement and the likes. These include using SQL injection or network probes to retrieve data from services running on different containers. Hence, by using this threat model, we keep our focus on just those movements and behaviours.

IV. HoneyKube Design

This research aims to devise a honeypot using the microservices architecture and deploy it on a cloud platform for data collection. The design uses the following objectives as guiding principles:

Fingerprintability: It is hard to distinguish the honeypot from a real system, i.e. network reconnaissance tools such as Nmap [27] and Shodan [31] have a tough time fingerprinting this system as a honeypot.

Interaction Level: It provides considerable interaction surfaces to engage the attackers and enables real data collection, i.e. that the honeypot is a medium-high interaction level honeypot.

Monitoring: It monitors and records interactions.

Security Defences: It deploys defences to prevent the misuse of the system by the adversaries. Recall from Section I that the higher the interaction level of the honeypot, the higher the risk of the adversaries taking over the system and exploiting it to harm other people, e.g. by using the system to send phishing emails or perform DDoS attacks.

A. Design Overview

We design a honeypot using the microservices architecture in K8s with a real-world application on top of it. Figure 2 shows a high-level picture of this design. The essence of a research honeypot is the monitoring and detection setup since that is responsible for recording the attack data. Recall that the microservices architecture adopts a distributed computing setup. Hence, we need to monitor the system from multiple observation points to ensure the logging of all interactions. To assist in the future analysis of attacker behaviours, we collect the data that can provide some crucial information about the attacks:

a) the source location of the attacker,
b) the vulnerability that provides the gateway into the cluster,
c) the first container that gets compromised, and most importantly,
d) the attackers' actions after entering the system, varying from lateral movement within the cluster to privileges escalation to gain access to the host systems.

The quality of data collected depends on the ability of the honeypot in deceiving the attackers. To increase the realistic factor of the honeypot, we took a real-world application codebase as a baseline. Employing a real-world application reduces the chances of fingerprinting the honeypot from its system settings, like seen in HoneyMix paper [16]. We also inject the honeypot with vulnerabilities to allow the initial breach and lateral movement within the cluster and increase the interaction level of the honeypot. Lastly, we employ security defences such as firewall and K8s security contexts when deploying the system. The firewall is placed at the network gateway to control the inbound and outbound traffic from the nodes. The K8s security measures enable us to restrict the traffic from the containers. These security contexts also help keep the attackers from escalating privileges and escaping the containers to the host systems.

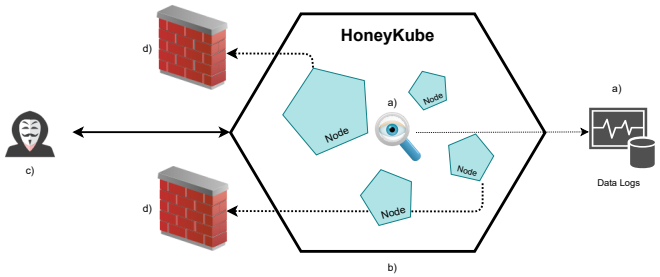


Fig. 2: **Overview of HoneyKube's design. The image depicts the high-level placement of all the major components of the honeypot: a) monitoring and recording, b) a honeypot using the microservices architecture and resembling a real system, c) open for attackers to exploit, and d) security defences in place to prevent misuse of the system.**

B. Design Details

With the monitoring infrastructure of the honeypot, we aim to collect data relevant for future analysis. To identify the origin of the attack, we need the source IP address of the attack, which requires capturing the incoming network traffic. Recall from Section II each microservice design focuses on a specific function and communicates with other services via HTTP or gRPC. In that case, attacks such as cross-site scripting and SQL injection can propagate within the cluster and potentially impact more than one microservice. Identifying the effect of these attacks on the microservices requires capturing the internal network traffic, i.e. the communication between the microservices. So, we design a monitoring setup to capture the inbound and outbound traffic from each microservice. The captured network traces give us a complete picture of the flow

of traffic within the cluster. Since the internet-facing interface of the application is a microservice as well, with this setup, we also capture the incoming network traffic from outside the cluster.

Next, we capture the actions taken by the attacker after entering the cluster by recording the system calls they execute. Recall that the microservices architecture consists of many containers running inside nodes. To get a complete set of actions performed by the attackers, we need to capture system call execution inside all the containers and the nodes within the cluster. Since the system calls are executed in the kernel space, and the containers share the kernel with the host machines, we set up the system call monitoring in all the nodes. In addition to the network traffic and the system calls, we also collect system logs from all computing units inside the cluster and audit logs from the K8s control plane.

The real-world application we use as a baseline for the honeypot is the open-source microservices demo application developed by Google Cloud Platform (GCP) [26]. This application provides a fully functional, deployment-ready imitation of an e-commerce web application. To keep the fingerprintability of the system low, we change the look and feel of the application and add new services and capabilities such that it does not resemble the demo application. With these changes, we transform an application with 11 microservices to an application with 14 microservices. Each of these microservices consists of a single container encapsulated in a pod. Hence, the application finally consists of 14 pods, each hosting a microservice.

Remember that a microservices architecture consists of multiple nodes, pods and containers, among other K8s objects. As the number of independent components in a system increase, so does the number of communication channels between them. With the increasing number of moving parts in a system, the attackers gain a larger attack surface to exploit. Hence, increasing the interaction level of the honeypot. To further increase the interaction level, we inject vulnerabilities into the system at multiple fronts. These vulnerabilities enable the attacker to breach the system, gain access to sensitive information and move laterally within the cluster.

Lastly, when conducting an experiment that requires the system to be vulnerable by design, it becomes our ethical and moral responsibility to ensure that it does not get used to cause harm to others. This harm can come in a comparatively moderate form as spam emails or a destructive form as a DDoS attack targeted towards innocent people. So, to prevent these kinds of mishaps, it is essential to incorporate a few security precautions in the design as one of the objectives. For this experiment, we restrict most of the outbound traffic from the cluster, only allowing it on the internet-facing frontend microservice of the application. Even on this microservice, we limit the traffic to the ports required by the microservice to function. By controlling the outbound traffic and limiting accessible ports, we block outgoing DDoS attacks and spam emails.

Since the microservices architecture employs a distributed setup, we require some additional precautions to ensure complete protection. Access to the control plane of the cluster can give an attacker administrative access to the entire system. Hence, we restrict the containers in the nodes from accessing the control plane by using Role-Based Access Control in K8s. Next, we remove root access from all the microservice containers, as with root access, an attacker can escape from the container and gain root access to the nodes. With that, the attacker can override the security measures and use the system for malicious purposes.

C. Challenges

Designing the monitoring setup for the microservices architecture comes with a few challenges. Here, we dive into these challenges and how we overcome them.

1) Network traffic capture: In the network monitoring setup, we capture the inbound and outbound traffic from each microservice. For this, we need to run a monitoring tool along with the microservice and capture the network traffic into (PCAP) trace files. The challenge we face in this scenario is monitoring the network traffic without the attacker detecting the presence of the tools or tampering with it in any way. Any process running within the microservice container can get easily detected by an attacker scanning for running processes in the container. Recall that each microservice container is running inside a pod, and a pod can hold multiple containers that share resources like the network stack. So, another container running in the same pod can capture the network traffic of the pod. We use this to isolate the application process and the monitoring process and run them in separate containers in a pod. Containers in a pod that provide supporting features to the main application container are called sidecar containers. With a sidecar container, we avoid any detection and tampering of the monitoring infrastructure by the attacker.

2) System Calls Capture: The multiple levels of abstraction in K8s architecture with containers running inside pods, which run on nodes, makes capturing system calls challenging. We need to capture system calls execution on all the levels to get a complete picture of the attackers' activities inside the cluster. Recall that the containers share the kernel with the host system, in this case, the nodes. Therefore, we need to run monitoring tools in each node to capture the executed system calls into trace files. The challenge in this design is that this will record all system calls and not just the anomalies, which, in this case, are system calls executed by the attackers. So, most of the collected data will be irrelevant, and identifying and tracking attackers actions in that dataset will be equivalent to finding a needle in a haystack. Also, the trace files containing system calls are large (e.g. 400MB in 3 minutes after compression). Considering the honeypot will stay active for a few weeks, this will pose a logistics problem. Extrapolating from 3 minutes to 1 week, we will collect approximately 1378 GB of trace files per week, which is not practically feasible for this experiment.

We solve these problems by devising a monitoring setup that uses a runtime security system that detects unexpected

behaviour as a trigger mechanism to identify when we need to capture the system calls. In this setup, we run the system calls capture into trace files on each node. The capture process uses a circular file rotation method to ensure continuous capture and contain the capture size. On a separate interface, we deploy the security system to monitor the cluster for anomalies. The security system raises alerts on detection. We use this alert as a trigger to notify the capture process on each node to permanently store the relevant files such that they do not get lost in the file rotation. These processes keep storing the files until they receive the notification to stop. With this, we record data relevant to attackers actions while containing the storage requirements.

V. Implementation

We can use any deployment platform that provides support for K8s as a cluster orchestrator to deploy the honeypot. For this research, we use the Google Kubernetes Engine (GKE).

A. Monitoring Setup

The monitoring infrastructure consists of two independent setups to collect two different types of data, network traffic and system calls. These mechanisms require different levels of access to the system, and hence, we place them at separate observation points within the cluster. In Figure 3 we see the placement of these mechanisms on the HoneyKube cluster.

In addition to the network traffic and system calls, we collect K8s audit logs and system logs from all the containers and the nodes in the cluster. K8s audit logs are the chronological record of calls made to the K8s API server. The control plane records all activities that involve users or applications sending requests to the K8s API. Container system logs are the logs a service generates during its lifecycle, based on the requests it receives from other services. These are the debug logs hard-coded into the application and meant to give insight into the functioning of the service. Node system logs are the Linux audit logs that provide detailed logging of security events on the system. The Linux systems generate these logs by default. These logs provide additional information about the attacks and their effects on the system.

1) Network Traffic Capture: For capturing the raw network packets, we employ the `tcpdump` command-line tool, which is a network packet capture utility. We use a `tcpdump` Docker image to set it up in a sidecar container in each pod. We configure these sidecar containers to start network capture automatically on pod creation. The `tcpdump` captures the network traffic into PCAP trace files inside the sidecar container. Recall that the pods are ephemeral in nature, which means we will lose any data or files stored on them upon restart. So, we define persistent volumes inside the pods, mount these volumes on the sidecar containers, and use them to preserve the captured trace files.

The sidecar containers do not communicate with any other container or service within the cluster. Hence, we assume that the attackers will not be able to identify their presence. But there still exists a remote possibility of an attacker identifying these containers and gaining access to them. In that case, they will have the ability to delete the collected trace files. So, we develop a file retrieval process that moves the files from the containers to the local storage, and to keep our losses to a minimum, we run it at regular intervals.

2) System Calls Capture: For capturing the system calls, we use Sysdig, an open-source tool that provides deep system visibility in Linux and container-based systems. Sysdig, a combination of several system-level monitoring tools such as `strace` and `htop`, enables capturing system calls into trace files. We set up the capture process with Sysdig on each node using a circular file rotation mechanism, such that at any moment, only `n` files are on the system. The names of the trace files end with the date and time of their creation.

We use Falco for the runtime system security to trigger the system calls capture. Falco is an open-source cloud-native runtime security tool that consumes kernel events, interprets them in runtime, and checks them against a list of pre-defined security rules. This enables Falco to detect suspicious activities and send out alerts in real-time. Falco operates by running on a separate pod on each node. The application containers inside these pods run with privileged access, giving them access to the kernel events of the host system. We refrain from using network channels such as HTTP/s or gRPC to send out Falco alerts to prevent detection by an attacker monitoring the internal network traffic of the cluster. The problem with an attacker intercepting this communication is that it gives away the local IPs of the Falco pods, using which the attacker can gain access to privileged containers in the cluster. This access can potentially facilitate escape from the containers to the host systems. Hence, we poll the standard output logs from all four Falco pods instead and look for alerts. We use the command-line tool `kubetail`, which allows continuous retrieval of logs from multiple pods at the same time. We set this up on our local system with a script that takes these logs as inputs and processes them to identify alerts.

When Falco raises an alert, the script detects it and triggers all the nodes. The capture process running on the nodes receives this trigger and enables the storage setting. Once this setting is enabled, these capture processes start storing a copy of the trace files to another location as soon as they finish writing to them to keep them from getting deleted by the file rotation system. The storing starts from the active file, the one that is currently capturing system calls. After a pre-configured period, the script sends another alert from the local system to the nodes disabling the storage setting.

B. Web Application

The real-world web application we use as a baseline imitates an e-commerce website, and all the microservices in it facilitate different features in the application. Hence, we keep the application as an e-commerce website while changing the product catalogue and revamping the frontend. The baseline application maintains its product catalogue in a text file. Since real applications favour storing their product catalogue in a database, we migrate it from a file to a MySQL database in

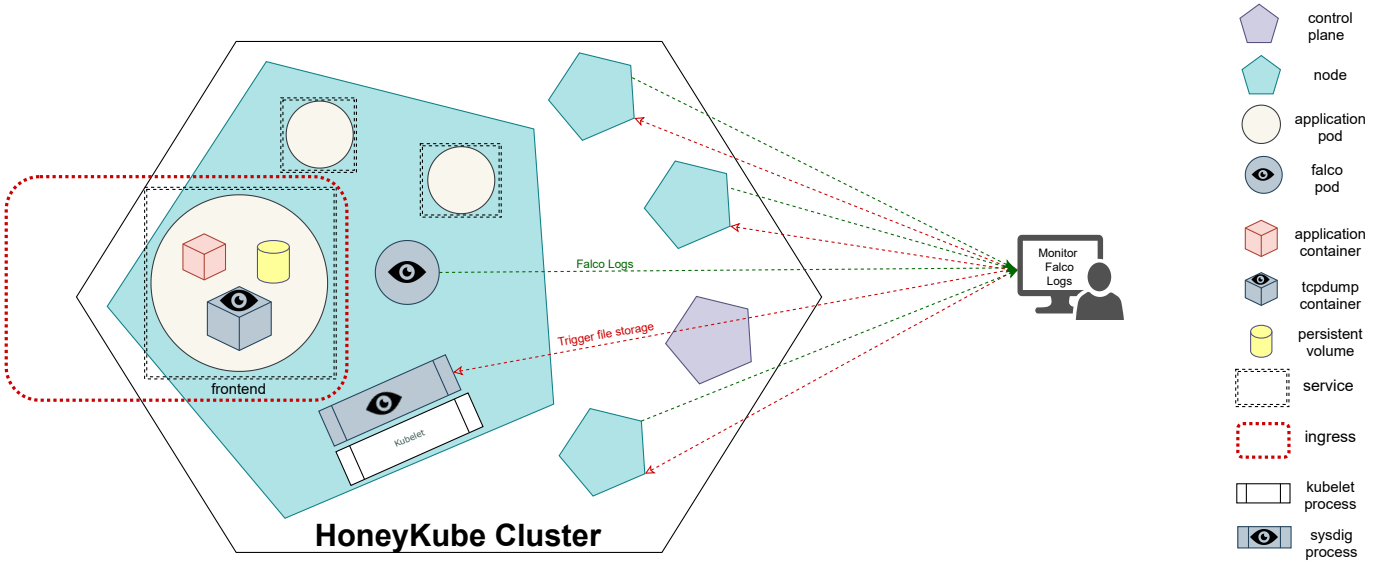


Fig. 3: **In-depth view of HoneyKube. We have a cluster with four nodes and a control plane. To capture the network traffic, we run tcpdump on a separate container in each pod, along with the application container. We use persistent volumes in the pods to prevent the loss of captured trace files. The system calls capture happens in two steps. First, we run the Sysdig capture process on the node machines using circular file rotation. Then we run Falco on a pod in each node to detect unexpected behaviour. We monitor the logs generated from Falco on a local system (as depicted here). When Falco identifies an intrusion, it raises an alert. The local machine uses this alert to send triggers to all nodes. This notifies the Sysdig processes to start permanently storing the captured trace files.**

a separate service to make the honeypot look believable. We also add features like user registration and user login in the application. To incorporate the functioning of these features, we add two microservices. First, a database service that runs a MySQL database and stores user account information. Second, an API service that runs a Flask API to access the user database. These features help make the application look genuine and in deceiving the attackers. The screenshots of the frontend of this application are in Appendix B.

C. Vulnerabilities

We increase the interaction level of the honeypot by injecting some vulnerabilities into it. These vulnerabilities facilitate the attackers in exploiting the system and consecutively improves the quality of the data collected from it. Using the K8s threat matrix [38], [39] developed by Azure Security Center, we map these vulnerabilities to the attack surface of HoneyKube. K8s threat matrix is similar to the MITRE ATT&CK framework[2], consisting of tactics and techniques used by attackers in a K8s environment. We inject vulnerabilities into the HoneyKube cluster to facilitate the attackers in exploiting the various stages of the threat matrix.

Initial Access. We allow initial access to HoneyKube by running a web application and exposing it to the internet. To make the application seem vulnerable to the attacker, we mishandle errors and print stack trace information on the user interface (UI). Even for basic errors like an incorrect password when attempting user login. With improper coding practices like this, we portray that the application consists of flaws and is exploitable. Figure 9 in Appendix B shows an example of this error trace.

We disclose some information to the attackers by printing them in the error trace [35]. This information includes the Golang version we use to develop the frontend microservice and the name of the user database (Appendix B, Figure 9).

Mapping these vulnerabilities to the K8s threat matrix [38], [39], we make use of the *Vulnerable application* technique to attract attackers.

Execution. We add vulnerabilities that allow the attackers to execute their code within the cluster. The frontend microservice runs the UI for the web application. We make this microservice vulnerable by using an older version (v1.11.5) of Golang instead of the latest one (v1.16.6) to develop it. There is a known CVE for this older version of Golang that makes it vulnerable to CRLF attacks [36]. CRLF refers to the Carriage Return (ASCII 13, \r) and Line Feed (ASCII 10, \n) characters. These characters note the termination of a line. A CRLF injection attack involves submitting a CRLF into an application by modifying either an HTTP parameter or the URL [37]. This attack can further escalate into more malicious attacks such as Cross-Site Scripting (XSS) and page

[2]https://attack.mitre.org/

injection attacks. By disclosing the Golang version in the error trace, we inform the attackers that the application is vulnerable to these attacks.

We accept the user inputs in the user registration and login forms on the UI without sanitizing them and forwards them to the user API microservice. This microservice receives the raw user inputs, directly concatenates them to SQL queries and propagates them further to the user database service. The service that hosts the user database executes these SQL queries and returns the result the same way. Hence, we make the application vulnerable to SQL injection attacks, enabling the attackers to move laterally within the cluster and access sensitive user data.

We establish an SSH server in the frontend microservice and configure it with easy-to-guess credentials. We use username `admin` for the credentials as attackers tend to aim for `root` or `admin` access on the system. The password is easily crackable by a brute-force attack. We also store this password in a text file in the `static` folder of the frontend after hashing it using MD5 and then encoding it using base64.

Mapping these vulnerabilities to the K8s threat matrix [38], [39], HoneyKube is vulnerable to *Application exploit* technique and the *SSH server running inside container* technique.

Credential Access. We add vulnerabilities that enable attackers in accessing credentials. In K8s, service accounts represent application identity. The configuration of each pod has an identity bound to it, and the pods use this identity to send requests to the K8s API. By default, K8s uses the same service account for all services and mounts its token on every pod. We create a new service account and give it authority to retrieve the secrets in the default namespace from the K8s API. The K8s secrets store sensitive information like credentials for the user database and service account tokens. So, by configuring the service account for the frontend service, we facilitate the attackers in accessing this information.

Mapping these vulnerabilities to the K8s threat matrix [38], [39], HoneyKube enables *List Kubernetes secrets* technique and the *Access container service account* technique.

Discovery and Lateral Movement. We aid the attackers in exploring the environment after entering the cluster. We configure the same service account we use to give attackers access to credentials to give the attackers the authority to query the list of services from the K8s API

The list of secrets that the attacker can retrieve includes the access token of the services accounts. In addition to the new service account, this also consists of the access token of the default service account that other services use. We increase the permissions of the default service account and allow it to retrieve information about the endpoints and services in the cluster. Additionally, we also give this service account the ability to update the services.

The scope of the containers is limited to the libraries and packages it requires for the application. Hence, we install command-line tools like `curl` and `wget` to the containers of some microservices to facilitate the attackers in their attack. With the help of these tools, the attackers can download tools to probe the network, increase their understanding of the environment, and move within the cluster.

Mapping these vulnerabilities to the K8s threat matrix [38], [39], HoneyKube enables discovery through *Access the Kubernetes API server* and the *Network mapping* techniques. While HoneyKube enables lateral movement through *Container service account* and *Cluster internal networking* techniques.

VI. Security Model

The higher the interaction level of the honeypot, the higher the amount of damage an attacker can inflict if they take over the system. Hence, it becomes essential to ensure the security of the system to prevent misuse by the attackers.

We add network policies in the K8s system to restrict all incoming traffic on all services. Then, we allow it only on the ports essential to keep the application functional, i.e. 22, 80 and 443. Similarly, we restrict all the outgoing traffic from all services and allow it on ports the frontend service uses, i.e. ports 80 and 443. Since the frontend service is the only service we expose to the internet, these policies enable us to restrict external access to other services. With these policies, we also restrict traffic on non-necessary ports such as the SMTP port and prevent the attackers from misusing the system for spam emails. Restrictions on traffic also reduce the chances of the system getting used for a DDoS attack.

Due to the large number of components that compose a K8s architecture, HoneyKube requires some additional defences to ensure adequate security of the system. As mentioned in Section III, we need to prevent the attackers from escaping the containers and gain access to the host system. *root* is the default user in containers. Hence, the containers have *root* access and privileges, which an attacker can use to gain *root* access to the host system (nodes). We prevent their escape by configuring the microservice containers to use a non-root user and remove all privileges. We configure all microservices with these settings in the Pod Security Context [43]. For these configurations, we set *privileged: false*, *runAsNonRoot: true*, and *allowPrivilegeEscalation: false*. These settings prevent a process from gaining more privilege than its parent process.

Next, we allocate fixed resources to each container to limit the amount of computing power they can use. We use a conservative strategy when allocating resources like CPU and memory (RAM) to prevent the attacker from using the containers to execute resource-intensive programs, such as crypto-mining and performing DDoS attacks.

The control plane in a K8s cluster has administrative control over all the resources inside it. Any attacker gaining access to the control plane can use these resources to launch cyber-attacks. Hence, it is essential to restrict access to the control plane from containers in the nodes. We enable the Role-Based Access Control in GKE to ensure limited access to the control plane from application containers.

We also configure the cluster to use shielded GKE nodes to prevent a known vulnerability in GKE [44]. In this vulnerability, an attacker can gain control over the entire cluster by exploiting the kubelet component [44]. With shielded nodes,

the control plane cryptographically verifies that each node is a valid virtual machine running in Google's data centre. As mentioned in the GKE documentation: "Without Shielded GKE nodes, an attacker can exploit a vulnerability in a pod to exfiltrate bootstrap credentials and impersonate nodes in your cluster, giving the attackers access to cluster secrets.". These secrets can provide administrator access to the GKE control plane from potentially any pod in the cluster. By enabling the shielded GKE node feature, we limit the attacker's ability to overtake the cluster by impersonating a node inside it.

Lastly, we add firewall policies to limit SSH port access on the nodes to only authorized IPs. We only allow the system we use for monitoring and data collection to access the node systems. These policies prevent attackers from gaining access to the nodes and escalating the privilege to get root access to the system.

VII. Experiments

For this research, we conduct two experiments: a) open experiment, where we expose HoneyKube to the internet, and b) controlled experiment, where we limit the exposure of HoneyKube to the IPs of the participants.

Environment. We use Google Kubernetes Engine (GKE) to deploy HoneyKube. The cluster configuration consists of 4 nodes and a control plane, where the node machines use E2 series high CPU machine that provides 4GB vCPU and 4GB memory along with Ubuntu as their OS instead of the Container-Optimized OS, which is the default option in GKE. It is essential to choose Ubuntu OS because it provides support for Sysdig in GKE nodes.

A. Ethical Practices

The deployment of honeypots comes with some ethical challenges. We are purposefully deploying a vulnerable system to lure attackers. In the attempt to look convincing, we may deceive some genuine users who believe the application is real. Additionally, the attackers can take over the system and use it for malicious purposes that harm innocent people.

The security defences we describe in Section VI prevents the attackers from taking over the system. We incorporate some features in the UI design of HoneyKube to discourage genuine users from using the application. Appendix A consists of the complete list of these features. For the controlled experiment, we request all the participants to sign a consent form (Appendix D).

We received approval from the ethics committee for both experiment settings (Reference number: RP 2021-131).

B. Open Experiment

In this experiment, we expose HoneyKube to the internet using HTTPS and collect data. To use the secure HTTPS protocol, we attach a TLS certificate and a domain to the service. Remember that in K8s, Ingress provides the method to expose services on HTTP/S protocols. We configure Ingress [45] to obtain a TLS certificate from the *Let's Encrypt* Certificate Authority(CA) and deploy the frontend service on the domain *https://techno.net.co*. Since the Ingress load balancer in GKE only allows exposure of HTTP/S ports, we expose the OpenSSH server on the frontend pod as a separate service. We add the IP of this service in the *robots.txt* file to leak it to the attackers.

C. Controlled Experiment

In this experiment, we restrict access to HoneyKube to the participants' IPs. Volunteers from the Twente Hacking Squad (THS) and the Red Team from Northwave, an intelligent security operations company in the Netherlands, participated in this experiment. There is no deception in this experiment, as the participants were aware that it is a honeypot. With this experiment, we simulate targeted attacks on the system. The HoneyKube settings in the two experiments are identical in terms of system architecture and design. Only one difference exists between the two systems, the method we use to expose the frontend service. Since there is no deception, we do not require a domain to make the application look believable. Knowing it is a honeypot, the participants will not enter sensitive information in the web application. So, there is no need for a CA issued TLS certificate. Hence, we expose the service directly on the internet, which allows us to keep the SSH port on the same IP address.

VIII. Evaluation and Results

This research aimed to design a honeypot using the microservices-based architecture and deploy it to collect attack data to enable future research in identifying attack patterns and attacker behaviour.

A. Datasets

We deployed the clusters from the two experiment settings on the GKE platform. The open experiment was active for two weeks and the controlled experiment for three weeks. We collected approximately 850 GB of data from the two experiments combined. The data is a mixture of system trace files, network trace files, and a combination of various types of log files. Below we describe the different types of data collected from these experiments.

1) System Trace Files: The system trace files (.scap files) generated by Sysdig comprise most of the collected dataset ($\sim$800 GB). We triggered the collection of these trace files whenever Falco raised an alert about an intrusion in the system. Falco gave the same warning when there was an attempt to SSH into the system and when there was a successful entry. Since the open honeypot witnessed recurring brute-force attacks, we collected a large number of these trace files. With these trace files, we captured the actions taken by the attackers, involving system calls, after breaching into the cluster. Hence, we succeeded in the challenge (mentioned in Section IV) of capturing the system calls with our monitoring setup and the chosen observation points.

2) Network Trace Files: The network trace files (.pcap files) were captured and stored in every pod of every microservice. From both the experiments combined, we collected approximately 8 GB of network trace files. With these trace files, we succeeded in capturing all the incoming and outgoing traffic from every microservice. Hence, our design succeeded in overcoming the network traffic capture challenge (mentioned in Section IV).

3) K8s Audit Logs: The K8s audit logs record all the requests made to the K8s API by users or by components that use the K8s API like the control plane. Each request to the K8s API generates an audit event at every stage of its execution. Hence the collected log files amount to around 40 GB of data. The logs are in JSON format, stored in text files. With this data, we can identify the effect of attackers' actions on the cluster as a whole.

4) Falco Logs: Falco, the tool used to detect intrusions or unexpected behaviour in the system and alerting the Sysdig capture process, stored all the events generated by it in log files. Since Falco is a security tool for cloud-native systems like K8s, its logs contain details about the components triggering the events like pod name and container image. These details help in giving a better view of the attackers' movements and actions inside the cluster. These logs are in JSON format, stored in text files.

5) System Logs: We group all the logs generated by all the containers and the nodes under this category. These are the default logs generated by the application containers that we configured to help with debugging. We also collected the records of all login attempts from the SSH server running on the fronted service. These logs consist of the username used in the login attempt and the source IP address. Additionally, nodes also contain audit logs generated by Linux systems that log all security-related events. All of these logs are in text file format. We keep all of these logs to give context to the attackers' behaviour identified by processing the collected data.

B. Fingerprintability

We used the Shodan Honeyscore tool to evaluate the fingerprintability of the HoneyKube. Shodan [31] is a search engine and crawler for devices connected to the internet and gathers in-depth information about these devices that traditional search engines do not, e.g. how many systems get affected by a new vulnerability. Shodan's developer API provides an endpoint, Honeyscore[3], that can fingerprint devices and identify whether they are honeypots or not. The Honeyscore takes an IP address as input and computes the probability that the host is a honeypot. The output value ranges from 0.0 (real machine) to 1.0 (honeypot).

The details behind the computation of Honeyscores are not publicly available. According to Shodan's creator, as mentioned in the HoneyPLC [10] paper, this tool uses the following criteria when scoring honeypots: a) a large number of open network ports, b) active service is not a match for the environment, e.g. ICS device running on GCP, c) markers from known honeypots like configuration settings, d) once a system is identified as a honeypot, it will most likely remain as a honeypot even after changing its configuration, e) a machine learning classification algorithm (not disclosed), and lastly, f) known honeypots use the same configuration.

The Shodan Honeyscore assigned a value of 0.0 to the HoneyKube honeypot, which means this tool could not distinguish between our honeypot and a real system. Hence it demonstrates that this state-of-the-art reconnaissance tool found it hard to fingerprint HoneyKube as a honeypot. Although, it is important to remember that, as of now, there are no known honeypots designed using the microservices architecture. Therefore, we can safely say that this HoneyScore tool has not yet been configured with fingerprintable markers to identify honeypots in this environment. So, even though Shodan HoneyScore does not provide a reliable platform to measure the fingerprintability of our honeypot, its evaluation does suggest that HoneyKube is not overtly identifiable as a honeypot.

C. Observations

We observed differences in the type of attacks witnessed by the two experiments.

1) Open Experiment: In the two weeks duration that the open experiment was active, we recorded roughly `11500` login attempts from more than `200` different IPs on the SSH server. These attempts originated from all around the world, from `36` different countries. The timestamps and the haphazard nature of these attempts suggest that these resulted from brute-force attacks executed by bots scanning the internet for vulnerabilities. In Table I we list the top five countries from which we recorded the maximum number of attempts. The comprehensive list of the countries is provided in Table II in the Appendix C.

Country	Number of Attempts
United States	2114
Vietnam	1946
Russian Federation	757
Pakistan	597
Albania	595

TABLE I: **Top countries and the corresponding number of login attempts recorded on the SSH server**

Through the SSH server logs, we recorded the usernames used for the brute-force attacks. We saw `36` different usernames used in these attacks. The most commonly observed usernames were `root`, `user`, `tech`, `demo`, and `telecomadmin`. The complete list of the recorded usernames is provided in Table III in the Appendix C.

Out of the roughly `11500` login attempts, only around `12` successfully logged into the SSH server. We observed that most of these disconnected immediately after successfully logging in to the server. But a few of them spent some time inside and performed some actions before disconnecting.

[3]https://honeyscore.shodan.io/

The captured system trace files recorded these actions. We observed varying types of behaviours through these system calls. One of them attempted to mine Monero bitcoin inside the container. One just tried to read and delete the system logs from the `/var/log` directory. A few others searched for active crypto mining processes inside the container by looking for the word "Miner" or "miner" in the list of running processes. Even though these behaviours varied, it is evident from the timestamps of the executed system calls that these were mostly automated attacks and did not have any manual intervention, all except one. The system calls captured from one of these attacks have a few seconds delay between the commands, as shown below. Hence there is a possibility that this attack involved manual intervention.

```
17:02:00 <NA>) ls -la
17:02:08 <NA>) sudo -i
17:02:12 <NA>) top
17:02:29 <NA>) ./.dhpcd -o 88.99.2xx.xx:80
17:02:37 <NA>) dmesg
17:02:37 <NA>) tail
17:02:40 <NA>) top -bn1
```

In addition to the attacks on the SSH server, we recorded brute force attempts to SSH into the node machines within the cluster. The `login` process in Linux logs all login attempts, whether they are successful or not, in separate files under the `/var/log` directory. `btmp` is a binary file that records all the unsuccessful login attempts and was part of the system logs we collected. The `btmp` files give us the list of usernames used during these login attempts and the IP addresses from where the attack originated. We observed that the attack attempts dated back to the creation of the cluster and originated from more than `4000` different IPs from `105` different countries. The comprehensive list is provided in Table IV in the Appendix C.

2) Controlled Experiment: With this setting, we simulated a targeted attack scenario. A total of 8 participants took part in this experiment: a) 5 from the Twente Hacking Squad (THS) and b) 3 from the Red Team from Northwave. Since the participants were all aware that the application was a honeypot, they approached it directly by searching for vulnerabilities. We recorded close to `1500` system calls executed by the participants in this setting. From the collected data, we observed that the participants used diverse tactics to breach the honeypot. These tactics varied from brute-force attacks on the SSH server to SQL injection on the frontend application.

We recorded breach attempts on multiple interfaces of the application. The logs from the frontend container show automated attempts to find hidden web pages in the application UI. Similarly, the container logs of the user API service show automated SQL injection attacks on the user database via the API service. The SSH server logs recorded brute-force attacks mainly with usernames:`root` and `admin`. These logs also recorded a few login attempts with credentials retrieved from the user database. The tactics used by the participants to compromise the honeypot are in line with the *Initial Access* and *Execution* tactics of the K8s threat matrix [38], [39].

We observed that even though different participants used different techniques once inside the cluster, the purpose of their actions was the same, to discover the environment they just entered. We recorded the download of tools like `nmap` [27], `kubectl` [28], `peirates` [29], and `linPEAS` [30] script to assist in the discovery process. Some attempted to map the internal network using `nmap`, while others used `kubectl` for this purpose. Some manually searched the directory structure and discovered the service account secrets mounted on the container. While others used `peirates`, a K8s penetration tool, to find those tokens. But, all of them used those tokens to query the K8s API and get the list of services and secrets. The K8s secrets contained the credentials for the MySQL server. After obtaining these credentials, we noted that most participants attempted to connect to the database server using several different ways.

D. Discussion of Results

Our research shows that a honeypot developed using the microservices architecture is invariably different from the regular monolithic ones. As discussed in the previous section, most of the attacks recorded in the open setting were automated attacks. These were most probably executed by bots scanning the internet for vulnerable systems to add to their botnet or mine bitcoin. None of the attacks we witnessed in the open setting targeted the microservices environment. Neither were they designed to adapt to the environment. Hence, the attack behaviour observed through the data collected in this setting does not vary from the monolithic systems.

The attack attempt, which we flagged as a possible manual attack, attempted to connect the SSH server container to a DHCP server. From the network interactions captured by our monitoring system, it appears it successfully established that connection. DHCP is the Dynamic Host Configuration Protocol. The `dhcpd` is a DHCP client that can configure a system to work on the DHCP server's network. We took down the open honeypot the day after this attack. So, there is no way for us to know how the attacker planned on using this connection.

The containers in a microservices architecture have a limited scope of what they can do. And with the security defences that we deployed on the honeypot, the automated attack scripts failed to gain root access. Hence, they were not successful in executing their attacks. An example of an unsuccessful attack is the one where there was an attempt to delete the system log files. One cannot execute commands to delete log files from the `/var/log` without root access. Similarly, the attempts to mine bitcoin failed due to the lack of root access and limited CPU and memory resources containers have at their disposal. And mining bitcoin is a resource-intensive process.

The data collected from the controlled setting provides a much better outlook into the different attack patterns required to exploit a microservices architecture. Even though the initial access points into the cluster were similar to those from the

monolithic system, the actions needed for further exploitation, such as privilege escalation, are remarkably different. Traditional tools, such as `linPEAS` are not as effective in finding vulnerabilities or ways to escalate privileges in this microservices environment. We observed this from the system calls captured from the controlled experiment. The participants using `linPEAS` switched to K8s specific methods of discovering the environment after running this script. From this, we infer that the traditional tools are not as effective in this environment. The K8s specific penetration tool, such as `peirates` were far more effective in obtaining credentials and enabling lateral movement within the cluster.

We also observed that the numerous components inside a microservices architecture are a liability in terms of security. In the open setting, this was evident from the attacks on the cluster nodes. Whereas in the controlled experiment, this became evident from the different interfaces used by the participants to breach the system. This research showed the large attack surface exposed in microservices architecture and the differences in the attacker behaviours when exploiting it. Understanding these differences is essential to developing tools to defend against attacks. Analyzing the data collected from this research can help improve our understanding of these differences and further improve the tools we develop to secure our system.

IX. Limitations and Future Work

The HoneyKube design proposed in this research contains a few limitations. In this section, we discuss these limitations and the different ways this research can move forward in the future.

1) Interaction Level. The quality of the collected data depends on the amount of interaction allowed to the attackers. These interactions are directly dependent on the vulnerabilities in the given environment that facilitate the attackers' movements. We injected vulnerabilities into HoneyKube to enable the attackers to perform various tactics, including but not limited to credential access, discovery and lateral movement. Even with the injected vulnerabilities, the attackers were allowed restricted infiltration into the system. Adding more vulnerabilities along the entire cyber kill chain will allow for higher mobility and a deeper infiltration by the attackers. And it will facilitate the collection of a much better quality of data.

2) Monitoring and Data Collection. The system calls capture mechanism we designed for HoneyKube requires improvement. Firstly, our current design will work if most of the actions taken by the attackers inside the containers trigger alerts in Falco. If that is not the case, it is possible to miss some activities inside the cluster. An example will be if the attackers stayed inactive after entering, and after some time, they become active and perform benign commands like `ls` or `ps`. In such a case, the activities will not get captured. Hence, we will not have a complete record of attacker activity within the system. Secondly, the trigger mechanism used to capture system calls can be more efficient by sending triggers only to the node experiencing activities instead of all the nodes. This way, we record system calls in just the node hosting the container where activities are detected and make the data collection mechanism even more efficient. Additionally, the implementation of the current design requires some improvements, as we observed some missing system trace files when the trigger mechanism failed. A more efficient method to trigger capture and storage of system calls will help improve the coverage of the collected data.

3) Experiment Duration. The types of witnessed attacks on the open honeypot indicate that mainly the automated bots discovered it, with only one of them possibly involving manual intervention. Hence, the honeypot did not reach its target audience, i.e. attackers. We can potentially remedy this problem by collecting the data for a longer duration. Or better yet, we can distribute the link to the honeypot across the dark web, similar to the factory honeypot [8] to attract more genuine attacks and facilitate data collection from targeted or manual attacks.

4) Future Work. Apart from improving the design to overcome the limitations of the current approach, as discussed above, the future of this research consists of identifying attackers' behaviour patterns. A thorough analysis of the collected data will assist in identifying attack patterns and tools used by the attackers in exploiting microservices architecture. We can cover more ground by repeating this experiment with different orchestrating platforms and different deployment platforms. The analysis of data from all of these experiments can assist in furthering our understanding of the attack surfaces in a generic microservices architecture, not tied to specific platforms. This analysis can further the research in devising better security systems to protect and defend microservices architectures.

X. Related Work

With microservices and K8s taking over the industry in a storm, the corresponding research to secure these systems needs to catch up. Hence, an increased requirement for new research to identify threats and vulnerabilities in these systems. Even though not much precedence exists in honeypots developed using microservices, there has been plenty of research towards new and innovative honeypots to meet the needs of the ever-evolving new technologies. In this section, we will dive into some of these honeypot designs. We will also look into some research that works towards designing security solutions for microservices-based systems.

There has been quite a lot of innovative research in developing security solutions to secure microservices and K8s. One such solution KubAnomaly [21] provides security monitoring capabilities for anomaly detection in a K8s cluster. KubAnomaly uses neural network approaches to create classification models that identify anomalies in the system. Another solution works towards making the microservices resilient to attacks by monitoring their resources usages [19]. Identifying abnormal usage, e.g. during a DDoS attack, marks the service as under attack and quarantines it on a reserve quarantine node in the cluster. Such that the availability of the service is not affected for the legitimate users.

Some of these solutions used sandboxing techniques to understand the threat landscape in this microservices environment. In 2018, researchers at Palo Alto Networks published their research on context-aware sandboxing techniques native to cloud environment [18]. Their work focuses on microservices and works towards improving our understanding of threats and attacks in this environment. In 2019, Sandnet [17] was proposed to enable the identification of threats and vulnerabilities in a microservices architecture by confining the microservice under attack in a sandbox network. Sandnet works by cloning the suspicious microservice, including all the microservices it communicates with, to the sandbox network and redirecting the traffic to the microservice in the sandbox network from the production network. A novel metric, Quality of Deception (QoD), is used to evaluate the network deception mechanisms proposed in this research. Even though this research comes close to what we tried to achieve with HoneyKube, it has a few limitations. Firstly, The design covers only a single point of entry to the cluster to identify intrusions, and it only handles one intrusion at a time. So, the presence of multiple adversaries at the same time is out of scope. Secondly, it only focuses on the network traffic behaviours of the attackers even after entering the cluster.

There has been rapid growth in the technology sector in the past few years. With new technologies, there is a rise in cyberattacks, with attackers targetting new and unidentified vulnerabilities. Trying to catch up with the ever-evolving technology, we see researchers designing innovative honeypots to identify attack patterns in these new systems.

Amidst the various cyberattacks on industries using Industrial Control Systems (ICS) like Stuxnet, Triton and WannaCry, researchers designed honeypots emulating ICS systems [8]–[13]. One such high interaction honeypot was the factory honeypot [8] developed by researchers from Trend Micro Research. They emulated a smart-factory solution for a fictitious company. To make it look believable, they added information such as a company backstory and employee contact details. They used real ICS hardware and a mixture of physical hosts and virtual machines. Once deployed, the honeypot was active for seven months. The honeypot recorded a large variety of threats, attacks, and a lot of fraudulent activities. These activities included cryptocurrency mining as part of a botnet, multiple ransomware attacks, fingerprinting attacks with scanners, and many control systems attacks on the PLCs in the system. This research confirmed that the more realistic a honeypot appears, the higher the probability of attackers taking the bait. This honeypot is an example of innovative honeypots designed to emulate a large and complex architecture. We also see the underlying risks of using high-interaction honeypots and the extent of monitoring and resources required to ensure their security.

Another innovative honeypot is the HoneyPLC honeypot. It focuses on developing high-interaction honeypots for Programmable Logic Controllers (PLCs) within ICS [10]. PLCs play a key role in bridging the cyber world and the physical world. They are responsible for controlling critical systems like centrifuge machines in a nuclear power plant. HoneyPLC proposed a solution to overcome the limitations of existing honeypot implementations for ICSs and PLCs [11]–[13]. These limitations include lack of complex functions, low interaction levels, easily fingerprintable as a honeypot and shallow implementation of network protocols. On evaluation with Shodan, the state-of-the-art reconnaissance tool, HoneyPLC received a Honeyscore of 0.0. Hence, confirming its covertness and ensuring that it is hard to differentiate between the honeypot and real PLCs. The use of this honeypot enabled the collection of meaningful data from their interactions with the adversaries.

Since honeypots are baits designed to catch attackers in action and monitor their activities on the system, they are effective as long as they manage to deceive the attacker. If not designed and configured correctly, these honeypots become vulnerable to fingerprinting attacks [40]. HoneyMix [16], software-defined networking (SDN) enabled intelligent honeynet, was proposed to mitigate such fingerprinting techniques. This honeynet makes use of SDN switches to forward the malicious traffic to the honeypots inside it. HoneyMix uses a Behaviour Learner to select the most desirable honeypot for the attack, and the incoming traffic is directed to that honeypot by SDN. HoneyMix also provided the ability to quarantine compromised honeypots using SDN switches and recover them using Network Function Visualization.

J.H. Jafarian and A. Niakanlahiji proposed "honeypots-as-a-service" (HaaS) [14], taking "software-as-a-service" as a baseline, to encourage small and medium enterprises to use honeypots for their security needs. Aside from providing a scalable and flexible plug-and-play service, they aimed to design an adequately deceptive honeypot that would be appealing to the attacker and indistinguishable from the production servers. They used a team of security experts to evaluate the generated honeypots.

XI. Conclusion

Our research proposed a novel honeypot design that uses microservices-based architecture. The objectives of the honeypot design were to monitor and record the interactions, be hard to fingerprint by reconnaissance tools, have an adequate interaction surface for the attackers to exploit, and secure the system against misuse by the attackers. To capture attacker interaction data, we devised an innovative monitoring setup. Due to the distributed nature of the microservices architecture, our monitoring setup records interactions from multiple observation points inside the cluster. We deployed a real-world application on top of our honeypot to make it hard to fingerprint and injected vulnerabilities into the system to increase the interaction surface. To protect the malicious use of our honeypot, we set up security defences and restricted the attackers' ability to take over. We deployed the final product and collected data in two separate experiment settings. From the collected data, we noted the differences in attack behaviours employed by the attackers when targeting the microservices-based system. We also observed how tools built

for traditional monolithic systems are not as effective when used with microservices. By extrapolating this observation to security tools, e.g. IDS, we deduce that we need specialized defences to secure microservices-based systems. Although we collected a large amount of data from our honeypot, there is room for improvement in design, implementation and deployment. And the collected data is of no use if it is not processed and analyzed to gather intelligence from it.

From the trends, it is visible that the microservices are here to stay and securing them is a challenge. We believe that our attempt in tackling this challenge is a step towards more secure microservices-based systems.

REFERENCES

[1] *Singer, G.*, Apr 2020. Threat Alert: Kinsing Malware Attacks Targeting Container Environments. [online] Available at: https://blog.aquasec.com/threat-alert-kinsing-malware-container-vulnerability [Accessed 16 July 2021].

[2] *Cimpanu, C.*, Sept 2020. Vast majority of cyber-attacks on cloud servers aim to mine cryptocurrency — ZDNet. [online] Available at: https://www.zdnet.com/article/vast-majority-of-cyber-attacks-on-cloud-servers-aim-to-mine-cryptocurrency/ [Accessed 16 July 2021].

[3] *Threatpost.com.* Jun 2021. Windows Container Malware Targets Kubernetes Clusters. [online] Available at: https://threatpost.com/windows-containers-malware-targets-kubernetes/166692/ [Accessed 16 July 2021].

[4] *Threatpost.com.* Apr 2020. Poorly Secured Docker Image Comes Under Rapid Attack. [online] Available at: https://threatpost.com/poorly-secured-docker-image-rapid-attack/154874/ [Accessed 16 July 2021].

[5] *Google Security Research.* 2021. CVE-2021-22555: Turning $\backslash x00\backslash x00$ into 10000$. [online] Available at: https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html [Accessed 16 July 2021].

[6] Gluck, A., 2020. Introducing Domain-Oriented Microservice Architecture. [online] Uber Engineering Blog. Available at:https://eng.uber.com/microservice-architecture/ [Accessed 10 August 2021].

[7] *L. Spitzner*. "The honeynet project: Trapping the hackers". IEEE Security and Privacy, vol. 1, no. 2, pp. 15–23, Mar. 2003. [Online].

[8] *Hilt, Stephan & Maggi, Federico & Perine, Charles & Remorin, Lord & Rösler, Martin & Vosseler, Rainer*. "Caught in the Act: Running a Realistic Factory Honeypot to Capture Real Threats". Jan. 2020

[9] *Ferretti, Pietro & Pogliani, Marcello & Zanero, Stefano*. "Characterizing Background Noise in ICS Traffic Through a Set of Low Interaction Honeypots". 51-61. 10.1145/3338499.3357361, Nov. 2019.

[10] *Morales, Efrén López & Rubio-Medrano, Carlos & Doupé, Adam & Shoshitaishvili, Yan & Wang, Ruoyu & Bao, Tiffany & Ahn, Gail-Joon*. "HoneyPLC: A Next-Generation Honeypotfor Industrial Control Systems". ACM SIGSAC Conference on Computer and Communications Security(CCS '20), Nov. 2020

[11] *Dániel István Buza, Ferenc Juhász, György Miru, Márk Félegyházi, and Tamás Holczer*. "CryPLH: Protecting smart energy systems from targeted attacks with a PLC honeypot." In Int. Workshop on Smart Grid Security. 2014. Springer, 181–192.

[12] *Stephen Hilt*, Github. "GitHub - sjhilt/GasPot: GasPot Released at Blackhat 2015". 2015. [Online]. Available: https://github.com/sjhilt/GasPot. [Accessed: 22- Aug- 2021].

[13] *Arthur Jicha, Mark Patton, and Hsinchun Chen.* "SCADA honeypots: An in-depth analysis of Conpot." In IEEE conference on intelligence and security informatics (ISI). 2016. IEEE, 196–198

[14] *Jafarian, Jafar Haadi & Niakanlahiji, Amirreza*. "Delivering Honeypots as a Service". DOI: 10.24251/HICSS.2020.227, Jan. 2020

[15] *Dowling, Seamus & Schukat, Michael & Barrett, Enda*. "New framework for adaptive and agile honeypots". ETRI Journal. https://doi.org/10.4218/etrij.2019-0155. July 2020.

[16] *Han, Wonkyu & Zhao, Ziming & Doupe, Adam & Ahn, Gail-Joon.*. "HoneyMix : Toward SDN-based intelligent honeynet". SDN-NFV Security 2016 - Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization, co-located with CODASPY 2016. Association for Computing Machinery, Inc, 2016. pp. 1-6, https://doi.org/10.1145/2876019.2876022.

[17] *Osman, Amr & Brückner, Pascal & Salah, Hani & Fitzek, Frank & Strufe, Thorsten & Fischer, Mathias & Tu*. "Sandnet: Towards High Quality of Deception in Container-Based Microservice Architectures". 10.1109/ICC.2019.8761171. IEEE International Conference on Communications (ICC) 2019: 1-7

[18] *Xu, Zhaoyan & Luo, Tongbo*. "Cloud-Native Sandboxes for Microservices: Understanding New Threats and Attacks". Blackhat Europe. December, 2018.

[19] *Ataollah Fatahi Baarzi, George Kesidis, Dan Fleck, and Angelos Stavrou.* "Microservices made attack-resilient using unsupervised service fissioning." In Proceedings of the 13th European workshop on Systems Security. EuroSec 2020. Association for Computing Machinery, New York, NY, USA, 31–36. DOI:https://doi.org/10.1145/3380786.3391395

[20] *Modi, Chirag & Patel, Dhiren & Borisaniya, Bhavesh & Patel, Hiren & Patel, Avi & Rajarajan, Muttukrishnan*."A survey of intrusion detection techniques in Cloud." Journal of Network and Computer Applications. 2013. 36. 42–57. 10.1016/j.jnca.2012.05.003.

[21] *Tien, Chin Wei & Huang, Tse-Yung & Tien, Chia-Wei & Huang, Ting-Chun & Kuo, Sy-Yen.* "KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches." Engineering Reports. 2019. 1. 10.1002/eng2.12080.

[22] *Dewa, Zibusiso & Maglaras, Leandros.* (2016)."Data Mining and Intrusion Detection Systems." International Journal of Advanced Computer Science and Applications. 7. 10.14569/IJACSA.2016.070109.

[23] *F. Salo, M. Injadat, A. B. Nassif, A. Shami and A. Essex*, "Data Mining Techniques in Intrusion Detection Systems: A Systematic Literature Review," in IEEE Access, vol. 6, pp. 56046-56058, 2018, doi: 10.1109/ACCESS.2018.2872784.

[24] *van Ede, T., Bortolameotti, R., Continella, A., Ren, J., Dubois, D. J., Lindorfer, M., Choffnes, D., van Steen, M. & Peter, A.* (2020, February). FlowPrint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic. In 2020 NDSS. The Internet Society.

[25] *van Ede, T., Aghakhani, H., Spahn, N., Bortolameotti, R., Cova, M., Continella, A., van Steen, M., Peter, A., Kruegel, C. & Vigna, G.* (2022, May). DeepCASE: Semi-Supervised Contextual Analysis of Security Events. In 2022 Proceedings of the IEEE Symposium on Security and Privacy (S&P). IEEE.

[26] *GitHub*. 2021. GoogleCloudPlatform/microservices-demo. [online] Available at: https://github.com/GoogleCloudPlatform/microservices-demo [Accessed 7 July 2021].

[27] *Gordon Fyodor Lyon*. 2009. Nmap network scanning: The official Nmap project guide to network discovery and security scanning. Insecure.

[28] *Kubernetes Documentation*. "kubectl". [Online]. Available: https://kubernetes.io/docs/reference/kubectl/. [Accessed: 23- Aug- 2021].

[29] *InGuardians, Inc.*, GitHub. "GitHub - inguardians/peirates: Peirates - Kubernetes Penetration Testing tool" [Online]. Available: https://github.com/inguardians/peirates. [Accessed: 23- Aug- 2021].

[30] *Carlos Polop*, GitHub. "PEASS-ng/linPEAS at master · carlospolop/PEASS-ng". [Online]. Available: https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS. [Accessed: 23- Aug- 2021].

[31] *John Matherly*. 2015. Complete guide to Shodan. Shodan, LLC (2016-02-25)

[32] *GitHub*. 2021. draios/sysdig. [online] Available at: https://github.com/draios/sysdig [Accessed 7 July 2021].

[33] *GitHub*. 2021. falcosecurity/falco. [online] Available at: https://github.com/falcosecurity/falco [Accessed 7 July 2021].

[34] *The OWASP Foundation*. 2010. "OWASP Secure Coding Practices Quick Reference Guide". Version 2.0.

[35] Academy, W., n.d. Information disclosure vulnerabilities — Web Security Academy. [online] Portswigger.net. Available at: https://portswigger.net/web-security/information-disclosure [Accessed 26 July 2021].

[36] *Cvedetails.com*. 2021. "CVE-2019-9741 : An issue was discovered in net/http in Go 1.11.5. CRLF injection is possible if the attacker controls a url parameter." [online] Available at: https://www.cvedetails.com/cve/CVE-2019-9741/ [Accessed 7 July 2021].

[37] *Owasp.org*. 2021. "CRLF Injection." OWASP. [online] Available at: https://owasp.org/www-community/vulnerabilities/CRLF_Injection [Accessed 7 July 2021].

[38] *Microsoft Security Blog*. "Threat matrix for Kubernetes." 2020. [online] Available at: https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/ [Accessed 16 August 2021].
[39] *Microsoft Security Blog.*. "Secure containerized environments with updated threat matrix for Kubernetes". 2021 [online] Available at: https://www.microsoft.com/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/ [Accessed 16 August 2021].
[40] *Desaster*. "Desaster/Kippo." GitHub, github.com/desaster/kippo.
[41] "Production-Grade Container Orchestration." Kubernetes, kubernetes.io/.
[42] "Ingress." Kubernetes, kubernetes.io/docs/concepts/services-networking/ingress/.
[43] "Kubernetes: Configure a Security Context for a Pod or Container." [online] Available at: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/ [Accessed 27 July 2021].
[44] *Wickenden, M.*, 2018. Hacking Kubelet on Google Kubernetes Engine. [online] 4ARMED Cloud Security Professional Services. Available at: https://www.4armed.com/blog/hacking-kubelet-on-gke/ [Accessed 27 July 2021].
[45] Google Cloud: Ingress for External HTTP(S) Load Balancing. [online] Available at: https://cloud.google.com/kubernetes-engine/docs/concepts/ingress-xlb [Accessed 27 July 2021].
[46] *Vasilomanolakis, Emmanouil & Karuppayah, Shankar & Kikiras, Panagiotis & Mühlhäuser, Max*. "A honeypot-driven cyber incident monitor: lessons learned and steps ahead". 10.1145/2799979.2799999, Sept. 2015
[47] *Van-Hau Pham & Marc Dacier*. "Honeypot Trace Forensics: The Observation Viewpoint Matters". 27 Future Generation Computer Systems. 539–546, June 2011.
[48] *Sergiu Eftimie*. "Honeypot System Based on Software Containers". doi: 10.21279/1454-864X-16-I2-062, 2017.
[49] *Safarik J et al.*. "Automatic Analysis of Attack Data from Distributed Honeypot Network". 8755 Proceedings of Spie - the International Society for Optical Engineering, May 2013
[50] *Nawrocki, M., M. Wählisch, T. Schmidt, C. Keil and Jochen Schönfelder*. "A Survey on Honeypot Software and Data Analysis". ArXiv abs/1608.06249, 2016.
[51] *Khattab, Sherif & Melhem, Rami & Mossé, Daniel & Znati, Taieb*. "Honeypot back-propagation for mitigating spoofing distributed Denial-of-Service attacks". Journal of Parallel and Distributed Computing - JPDC. 66. 10.1109/IPDPS.2006.1639674, Mar. 2006.
[52] *Fraunholz, Daniel & Zimmermann, Marc & Schotten, Hans*. "An adaptive honeypot configuration, deployment and maintenance strategy". 53-57. 10.23919/ICACT.2017.7890056. 19th International Conference on Advanced Communication Technology (ICACT) 2017
[53] *Pauna, Adrian & Bica, Ion*. . "RASSH - Reinforced adaptive SSH honeypot". IEEE International Conference on Communications. 1-6. 10.1109/ICComm.2014.6866707. 2014

APPENDIX A
STEPS TO PROTECT GENUINE USERS

We incorporate features in the UI design for the honeypot to protect any genuine users from getting deceived by the honeypot.

1) We add an "out of stock" banner to all product descriptions to dissuade legitimate users from trying to place an order. Figure 7 gives an example of this.
2) The user sign-up form only requires a username and password, and it does not ask for any personal information. Figure 5 shows the registration page.
3) The web server uses a secure HTTPS connection to ensure no sensitive data can be extracted by monitoring the network traffic.
4) The order checkout form does take credit card information, with an option to save the credit card number for future use. As this is sensitive information, the HTTPS connection between the client-side UI to the server will ensure the security of the data. The credit card numbers get replaced with random values upon being received by the server, ensuring that all the processes, from here on, use this randomly generated number. Hence, the card validation function flags every card number as invalid. And every order placement request is rejected. If the "save for future" option is selected, this randomly generated credit card value gets stored in the database. This mechanism makes the honeypot realistic without actually storing any sensitive data.
5) Other personal information like email address, home address, pin-code, city, state, country and CVV (for credit card) the user might enter in the checkout form will also get replaced with dummy values upon being received by the web server. None of the personal information gets stored in the database, and replacing these values with dummy values ensures that this data does not appear in the application logs or the network traffic between the microservices.
6) The monitoring setup of the honeypot will capture the user's IP address and store them in trace files and log files. For the honeypot, it is essential to identify the location of the incoming traffic, for which we require the IP address. Hence, within a week of recording the data, we process it, and anonymize the IP addresses. The IP addresses in the network trace files will be hashed and rewritten to the PCAP files, obscuring the IP addresses and preserving the TCP flows.

APPENDIX B
USER INTERFACE

For HoneyKube's frontend application, we replicate an e-commerce website, selling electronic devices like smartphones and smartwatches. The various web pages from the application are given below.

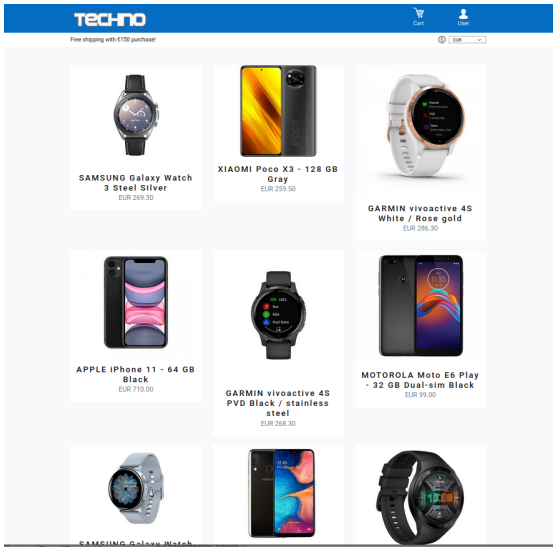


Fig. 4: **HoneyKube application UI: Home page and the Product Catalogue**

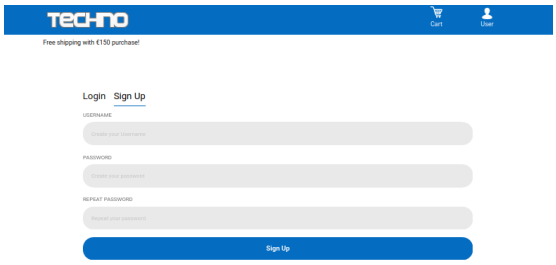


Fig. 5: **HoneyKube application UI: Login/Signup Page**

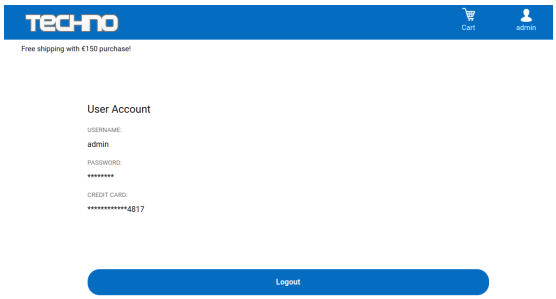


Fig. 6: **HoneyKube application UI: User Account Details**

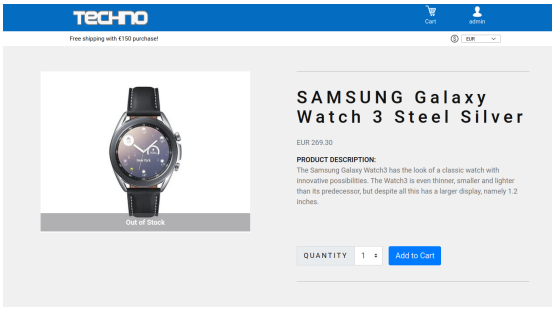


Fig. 7: **HoneyKube application UI: An example product description page showcasing the "out of stock" banner.**

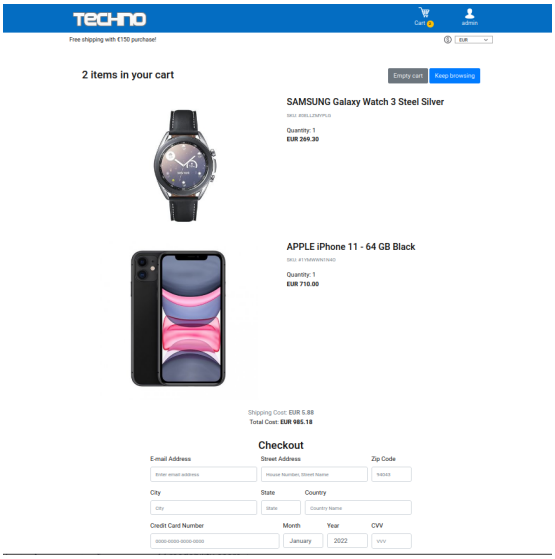


Fig. 8: **HoneyKube application UI: Shopping Cart**

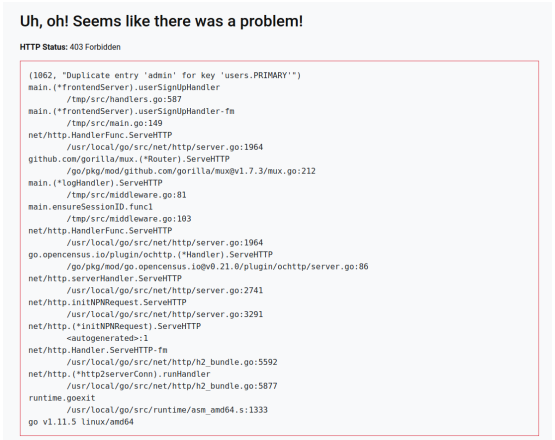


Fig. 9: **HoneyKube application UI: Error trace and an example of the Information Disclosure Vulnerability. Here we can see the Golang version printed on the last line and the name of the user database printed on top.**

APPENDIX C
DATA COLLECTED BY OPEN EXPERIMENT

A. Brute-force Attacks on SSH server

The brute-force attacks recorded on the SSH server originated from 36 different countries. We list these countries, along with the number of login attempts that we recorded originating from that country.

Country	Number of Attempts
United States	2114
Vietnam	1946
Russian Federation	757
Pakistan	597
Albania	595
Indonesia	595
China	130
Panama	106
Thailand	96
Canada	72
Ecuador	62
Germany	36
Netherlands	36
Egypt	34
France	32
Italy	20
Poland	18
Korea, Republic of	16
South Africa	14
Sweden	14
Australia	12
United Kingdom	10
Greece	8
Tunisia	8
Lithuania	8
India	6
Argentina	6
Moldova, Republic of	4
Ukraine	4
Philippines	4
Colombia	4
Hungary	4
Denmark	4
Brazil	4
Bulgaria	2
Switzerland	2

TABLE II: Complete list of countries the attacks originated from

These brute-force login attempts used various usernames. We list these usernames below in the order from most commonly recorded usernames to least commonly recorded username.

Usernames
root
user
tech
demo
telecomadmin
profile1
admin1
support
default
ubnt
web
user1
administrator
MikroTik
pi
admin
oracle
usuario
test
ftpuser
test1
test2
fixed
contador
a
duni
baikal
postgresadm
hadoop
a1
a2
a3
aa
jenkins
uploader
sysdbadm

TABLE III: Complete list of usernames used in brute-force attack

B. Failed login attempts on nodes

The nodes in our honeypot are virtual machines using Ubuntu OS. The `login` process in Linux records the failed login attempts in a binary file called *btmp*. We processed those *btmp* files from all the nodes and compiled the list of countries from where those login attempts originated.

Country	Number of Attempts
Vietnam	17680
Indonesia	10389
Thailand	9464
United States	8826
India	5260
Germany	4883
Russian Federation	4834
Brazil	4329
China	2560
Malaysia	2009
Egypt	1981
Colombia	1924
Peru	1666
Romania	1652
Pakistan	1649
Mexico	1563
Turkey	1480
Argentina	1379
Kazakhstan	1206
Netherlands	1186
Philippines	1172
Dominican Republic	1119
Ukraine	864
Venezuela, Bolivarian Republic of	792
Canada	735
Korea, Republic of	726
Poland	644
Spain	594
United Arab Emirates	593
Greece	580
Portugal	574
Saudi Arabia	559
Georgia	554
Guatemala	552
Italy	493
Morocco	386
United Kingdom	384
Kenya	373
Australia	366
Nicaragua	359
Albania	353
Lao People's Democratic Republic	342
Zimbabwe	330
Belgium	315
Serbia	300
Hungary	289
Chile	289
Belarus	284
Bangladesh	283
New Zealand	281
Mongolia	281
Costa Rica	281
Bahrain	279
Kyrgyzstan	278
Senegal	278
Nepal	269
Taiwan, Province of China	267
Ecuador	266
Singapore	250
Panama	248
Trinidad and Tobago	228
Algeria	197
Czech Republic	146
France	121
Monaco	80
Sweden	70
Lithuania	57
Japan	44
South Africa	29
Switzerland	19
Tunisia	15
Hong Kong	13
Denmark	13
Norway	13
Austria	11
Croatia	9
Paraguay	7
Cambodia	7
Kuwait	6
Ireland	5
Finland	5
Curacao	4
Uruguay	4
Israel	4
Moldova, Republic of	4
Sri Lanka	3
Bosnia and Herzegovina	3
Palestine, State of	2
Nigeria	2
Iceland	2
Jordan	2
Comoros	2
Malta	2
Cayman Islands	2
Rwanda	1
Uzbekistan	1
Slovenia	1
Uganda	1
Bolivia, Plurinational State of	1

Country	Number of Attempts
Madagascar	1
Tanzania, United Republic of	1
Ghana	1
Papua New Guinea	1
Azerbaijan	1
Armenia	1
Botswana	1
Ethiopia	1
Honduras	1
Zambia	1
Belize	1
Bulgaria	1
Mozambique	1

TABLE IV: Complete list of countries from where the attacks on the nodes originated

Appendix D
Consent Form

Honeykube

This brochure provides information about the HoneyKube experiment.

1.1 General Information

Dear reader,

On this page, we would like to inform you about the HoneyKube experiment that you have agreed to participate in. In this research project titled "Honeykube: Design and development of honeypot for micro-services based cloud architecture", a honeypot emulating a micro-services based web application will get deployed on the cloud, and all of the activities on it will be logged and analyzed. As honeypots are decoy computer resources whose value lies in being probed, attacked, or compromised, this system will not have any production value. This research aims to collect data from all the interactions with the honeypot. Analyzing this data will provide insight into the attacker's behaviour when targeting the systems developed using this architecture and assist further research in designing adequate defences to secure microservices.

There are some important aspects that you, as participants, should be aware of before taking part in the experiment. First, the web application (honeypot) you will be interacting with is a dummy e-commerce website that supposedly sells electronic products like smartphones and smartwatches. The website emulates a realistic look and feel to increase the chances of deceiving a malicious user into believing it is a real application. Therefore, it consists of features such as "add to cart", "user login", "cart checkout", and "payment service". You are requested to not enter any sensitive or personally identifiable information (PII) on the website when trying to breach it as that data gets stored in insecure databases. Second, all the network traffic to and from the application and between the microservices is captured for further analysis. This might contain your IP address. Since, for the honeypot, it is essential to identify the location of the incoming traffic, the data will be processed within a week of it getting recorded. After that, the IP addresses will be randomized and rewritten in the saved PCAP files, anonymizing the IP addresses in compliance with the GDPR. Finally, participation is voluntary, and you can decide to stop at any step of the experiment without giving any reason. Other relevant aspects are that your data will be handled confidentially. We do everything in our power to anonymize your data. Should you consent to participate in this research, only an anonymized version of the dataset may be shared with other researchers.

This experiment will be active for 2 weeks, and you are free to participate at any point within that period.

To participate in this research, the following pre-requisites apply:

- Must be an adult (18 years or older).

The research does not offer any remuneration to participants.

Yours sincerely,
Researcher: Chakshu Gupta, MSc Computer Science, University of Twente. Email: c.gupta@student.utwente.nl
Supervisor: Dr.ir. Andrea Continella. Email: a.continella@utwente.nl
Supervisor: Thijs van Ede. Email: t.s.vanede@utwente.nl

Please contact Chakshu Gupta (c.gupta@student.utwente.nl) for any questions regarding the experiment.

If you have any complaints about this research, please direct them to the secretary of the Computer Information Science Ethics Committee of the Faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente, P.O. Box 217, 7500 AE Enschede (NL), email: ethicscommittee-cis@utwente.nl).

1.2 Consent Form

I hereby declare that I have been informed in a manner which is clear to me about the nature and method of the research as described in the aforementioned information brochure (this document). My questions have been answered to my satisfaction. I agree of my own free will to participate in this research. I reserve the right to withdraw this consent without the need to give any reason and I am aware that I may withdraw from the experiment at any time. If my research results are to be used in scientific publications or made public in any other manner, then they will be made completely anonymous. My personal data will not be disclosed to third parties without my express permission. If I request further information about the research, now or in the future, I may contact Chakshu Gupta, via email (c.gupta@student.utwente.nl).

Signed in duplicate:

.....	.....
Name subject	Signature