HoneyKube: Designing and Deploying a Microservices-based Web Honeypot

Chakshu Gupta
University of Twente,
The Netherlands
c.gupta@utwente.nl

Thijs van Ede
University of Twente,
The Netherlands
t.s.vanede@utwente.nl

Andrea Continella
University of Twente,
The Netherlands
a.continella@utwente.nl

***Abstract*—Over the past few years, we have witnessed a radical change in the architectures and infrastructures of web applications. Traditional monolithic systems are nowadays getting replaced by microservices-based architectures, which have become the natural choice for web application development due to portability, scalability, and ease of deployment. At the same time, due to its popularity, this architecture is now the target of specific cyberattacks. In the past, honeypots have been demonstrated to be valuable tools for collecting real-world attack data and understanding the methods that attackers adopt. However, to the best of our knowledge, there are no existing honeypots based on microservices architectures, which introduce new and different characteristics in the infrastructure. In this paper, we propose HONEYKUBE, a novel honeypot design that employs the microservices architecture for a web application. To address the challenges introduced by the highly dynamic nature of this architecture, we design an effective and scalable monitoring system that builds on top of the well-known Kubernetes orchestrator. We deploy our honeypot and collect approximately 850 GB of network and system data through our experiments. We also evaluate the fingerprintability of HONEYKUBE using a state-of-the-art reconnaissance tool. We will release our data and source code to facilitate more research in this field.**

***Index Terms*—Microservices, Kubernetes, Honeypots, Web applications, Web security**

I. INTRODUCTION

Microservices architecture evolved from the traditional Service-Oriented Architecture (SOA) and, in recent years, became the de-facto standard for developing large web applications. The change in the business landscape post-COVID-19 has boosted the global popularity of microservices for cloud applications. The recent report on the "Global Cloud Microservices Industry" predicted the global market for cloud microservices to reach $4.1 billion by 2030 from $981.6 million in 2022 [1]. Tech giants like Amazon, Netflix, Spotify, and Uber have also contributed to the increase in popularity of microservices by adopting and endorsing them [2], [3]. The modular paradigm of microservices dictates that applications are decomposed into loosely coupled, independent parts, each running within an isolated container. There are several benefits to this modular architecture, including scalability, faster deployment, and security benefits like the separation of concerns. However, its decentralized nature poses fundamental challenges in fortifying the security of these environments. With the increased adoption of microservices, there is also a rise in the motivation amongst attackers to find innovative methods to compromise them. In the last few years, we have witnessed numerous attacks against such containerized systems [4]–[11]. These attacks ranged from infected docker images pushed to Docker Hub [9] to the development of malware to break out of containers and establish a backdoor [8]. Because of the fundamental differences in the underlying technology, attacks targeting microservices-based web applications differ from traditional attacks designed for monolithic applications. Since these differences exist in methods and targeted vulnerabilities, traditional intrusion detection and prevention systems (designed for monolithic systems) are less effective when applied in containerized environments [10], [11]. Hence, to properly understand the differences in attack patterns and design effective security solutions, we need real-world data about the methods adopted by attackers when infiltrating microservices-based web applications.

Honeypots are one of the main tools for gathering such real-world data. The honeynet project[1] developed honeypots to facilitate data collection from genuine attacks and use that data to identify attack patterns. Spitzner defined honeypots as "decoy computer resources whose value lies in being probed, attacked, or compromised" [12]. Since decoy systems have no production value, any access attempt or interaction with these systems is considered a probe, scan, or attack. All the system activities are logged and analyzed to better understand attackers' behavior. The insights gained from honeypots have been vital in advancing security defenses in academic and industrial settings [13]–[25].

The amount and the quality of the data collected from the honeypot depend on a) the level of interaction permitted to the user in the system—low, medium, and high interaction honeypots—and b) 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. Due to the limited interaction, they capture a limited amount of information about the attackers' actions. Whereas high-interaction honeypots are real systems, imitating production systems with open vulnerabilities. Although the higher interaction in these honeypots provides more insights into the attacks, they pose a greater risk of getting used for real malicious campaigns, e.g., as part of a

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

botnet. Thus, they are quite expensive to deploy and maintain.

In this research, we design a new web honeypot, HONEYKUBE, using the microservices architecture. We build HONEYKUBE as a realistic web application and expose it to the Internet to collect attack data. Our application uses Kubernetes for container orchestration and Google Kubernetes Engine (GKE) for deployment. The monitoring system design of this honeypot consists of capturing interaction activities at multiple observation points to accommodate the modular conditions of microservices. HONEYKUBE collects the system calls executed within each container and all network interactions with the honeypot, including internal flows between the microservices.

We deployed HONEYKUBE for one month and collected nearly 850 GB of data consisting of system trace files, network traces, and additional log files.

In short, our paper makes the following contributions:

- We present a scalable monitoring and detection technique to record the runtime activity of microservices-based web applications.
- We leverage our monitoring framework to design and implement HONEYKUBE, the first web honeypot based on a microservices architecture.
- We collect and release a dataset of real-world attacks to web-based containerized environments. This data will foster future research in understanding the attackers' behavior and attack patterns.

In the spirit of open science, our source code and data are publicly available at https://github.com/utwente-scs/honeykube.

II. BACKGROUND

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

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) and API calls. Unlike a single database in monolithic architecture, every microservice can 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. Containers share the kernel with the host machine and use kernel features such as 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.

Kubernetes. Building large, complex applications such as Amazon and Uber require many microservices to support all features. For instance, Uber's application uses more than 2,000 microservices [26]. We require orchestration tools such as Kubernetes (k8s) [27] to automate the monitoring and management of these microservices. A cluster in k8s is a distributed computing setup consisting of a set of worker machines called *nodes*. A *node* is an abstraction of a machine and can be either physical or virtual. Figure 1 assists in visualizing the K8s cluster and its components. The microservices run on the nodes. The *control plane* is the cluster orchestrator handling the scaling and scheduling of the microservices. The core of the control plane is the *K8s API server* that enables its' communication with the nodes to ensure their health and manage their state. Each node includes a running agent *kubelet* for this communication.

K8s employs an additional abstraction to group one or more containers, sharing network and storage resources, called a *pod*. Hence, the containers in each pod can communicate with each other using `localhost`. A pod is an ephemeral resource and loses its state upon restart. Any data requiring persistence should use persistent volumes within the pods for storage. Each pod is assigned a unique private IP address, which changes with every restart. The pods remain isolated by default until they are configured to enable communication (internal or external) to the cluster. A logical set of pods that work together to fulfill one design requirement can be configured together as a microservice, along with the access policy using a *service*. *Service* is an abstract method to expose the application running a set of *pods* as a network service [28]. The IP address assigned to a *service* does not change throughout its existence. An API object, called *ingress*, exposes these services to the outside world. *Ingress* provides load balancing, SSL termination, and name-based virtual hosting capabilities [29]. With *Ingress*, we can expose HTTP and HTTPS routes from the Internet to services inside the cluster. Figure 1 provides a visual depiction of the relation between Ingress, Service, and the pods within the cluster.

III. THREAT MODEL

Our goal is to monitor the runtime activity of a microservices-based web application to record the effects of cyber threats on the honeypot. We assume that the attackers aim to escalate privileges to gain access to the host systems from the containers. We set up security defenses to limit the attack surface by ensuring the containers run as unprivileged users with limited capabilities. We describe these defenses in detail in Section VI. We assume that the attackers cannot gain access to other cloud resources from our Google Cloud Platform (GCP) account [30], [31]. We also assume that the attackers cannot shut down the experiment and use the resources for malicious purposes. We base our assumption on the strictness of the default Identity and Access Management (IAM) in GCP. Every compute engine uses a service account to define its access identity. We use the default service account that is configured with limited access. Our final assumption is that the attackers cannot discover or tamper with HONEYKUBE's monitoring system to evade detection since we use resources within the cluster to store the monitoring data. It is a common assumption in threat monitoring systems [32],

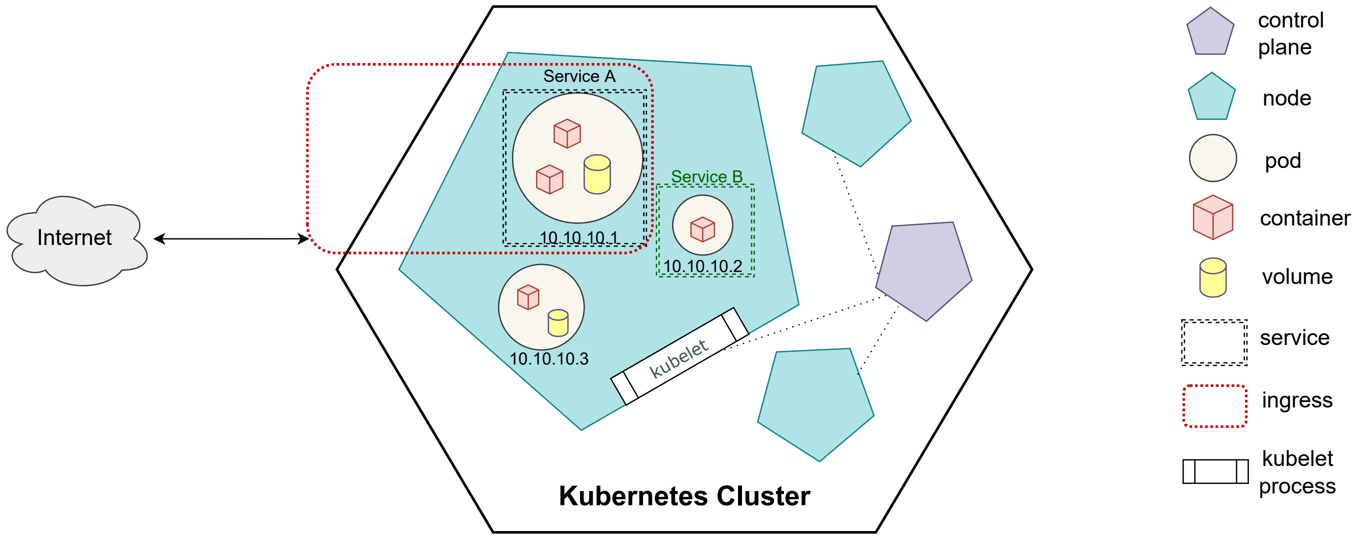

Fig. 1: **Detailed architecture of a K8s cluster showing its various components.** The cluster is a combination of nodes and a control plane. A Node is an abstraction for a machine. The control plane is the cluster orchestrator. The dotted lines between them depict the communication channels between 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.

[33]. Despite the assumption, we take some precautions (as we mention in Section V) to minimize the damage in the case of a breach.

IV. HoneyKube Design

We aim to devise a web honeypot using microservices to collect data about cyberattacks. Naturally, honeypots are designed as baits for attackers. So, the more realistic the trap, the higher the chances of the attackers biting. To this end, we set the following objectives for our design:

Fingerprintability: If the attackers realize that the attacked system is a honeypot, they will not attempt to compromise it. Hence, it should be hard to distinguish the honeypot from a real system. Network reconnaissance tools, such as Nmap [34] and Shodan [35], should find it hard to fingerprint it as a honeypot.

Interaction Level: The honeypot should provide considerable interaction surfaces to engage the attackers to enable real-world data collection.

Monitoring: The honeypot should be able to effectively and efficiently monitor and record the activities performed by the attackers.

Security Measures: While the honeypot is intentionally designed to be exploitable, we need to prevent attackers from causing damage to others on the Internet. Therefore, it is essential to introduce security measures to prevent misuse, such as sending phishing emails or performing DDoS attacks.

A. *Design Overview*

HoneyKube's design focuses on fulfilling the requirements stated above. The monitoring setup is the most essential part of a honeypot since it is responsible for the collection of data from the attackers' interactions. The architecture consists of multiple microservices, communicating both internally with each other and externally with the rest of the Internet. Therefore, our honeypot monitors runtime activity at four levels: a) **External network traffic**, for identifying reconnaissance attacks, initial access, command & control, and potential data exfiltration; b) **Internal network traffic**, for tracking lateral movement and potential privilege escalation; c) **System calls**, for reconstructing the attackers actions within containers and nodes; d) **System logs**: for tracking and correlating the propagation of attacks throughout the rest of the cluster in the form of node failure, container crashes, etc.

We use an open-source application as a baseline to build a realistic web-based application on top of our monitoring infrastructure. Employing a real-world application reduces the chances of fingerprinting the honeypot from its system settings, as shown in previous research [36]. Hence, this increases the chances of the attackers taking the bait. At the same time, we introduce significant changes to the original open-source application. Then, we enlarge the playing ground for the attackers by injecting vulnerabilities in the honeypot. These vulnerabilities allow attackers to breach the system and move laterally within the cluster, improving the quality of the data we collect. Lastly, we try to minimize the damage an attacker can cause if they gain complete control of the system. We employ measures to restrict the traffic from the containers and the nodes and set up security contexts to prevent privilege escalation and escape from the containers to the host system.

B. *Design Details*

Monitoring. We design a monitoring setup to capture all inbound and outbound network traffic from each microservice. As the Internet-facing interface is also a microservice, this setup captures the external network traffic to and from the cluster, along with the internal network traffic. Next, we capture all system calls executed inside each container and node within the cluster to reconstruct all the performed actions after gaining initial access. Since the system calls are executed in the kernel space, and the containers share the kernel with the host machines, i.e., the nodes, we set up syscall capture in the nodes. Finally, we collect logs from all computing units within the cluster, including the K8s control plane.

Fingerprintability. We use a deployment-ready open-source web application designed using the microservices architecture as a baseline for our honeypot to give it a realistic look and feel. This application, developed by Google Cloud Platform (GCP), imitates an e-commerce website [37]. Since this induces the risk of the attackers being familiar with the demo website and hence identifying it, we change the original layout of the application and add microservices with new functionalities. Each of these microservices consists of a single container encapsulated in a pod. The final application consists of 14 pods, each hosting a microservice.

Interaction Level. A honeypot requires a large interaction surface to keep the attackers engaged and facilitate the collection of good-quality data. Remember that a microservices architecture consists of multiple nodes, pods, and containers, among other K8s objects. Our application's exploitable attack surface contains 14 microservices spread across 4 nodes and 14 pods. Additionally, we add vulnerabilities to these microservices to increase their exploitability and facilitate the attackers during their malicious activities. This includes breaching the system, gaining access to (fake) sensitive information, and moving laterally within the cluster.

Security Measures. We employ security measures on the honeypot to minimize the impact of the attacks on other machines. We limit the network activity of the honeypot using security contexts in K8s and restrict all unnecessary outbound traffic to block outgoing DDoS attacks and spam emails. We prevent the attackers from escaping the containers by reducing the privileges from the containers and using Role-Based Access Control. The details of all the security measures are given in Section VI. While these measures reduce the interaction surface, they increase the realism of the honeypot since real production-level applications have similar security measures in place to protect them.

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.

Hiding Monitoring Setup. Attackers should be able to interact with the honeypot without detecting that they are being observed. Capturing the inbound and outbound traffic from each microservice requires running a monitoring tool along with every microservice. Such a process can be easily detected by scanning for the active processes from within the container. Therefore, we use sidecar containers (a container providing supporting features to the main application container within the same pod) to capture and store all network traffic to and from that pod. With sidecar containers, we hide the monitoring process from the application process and avoid immediate detection and tampering by attackers.

Capturing Attack-Related Syscalls. Due to the distributed nature of the microservices architecture, capturing system calls from all the system's components is already expensive. Since we capture system calls at the kernel level (of nodes), we also capture system calls generated by our HoneyKube, in addition to the ones made by attackers. However, the analysis of attacks requires only attack-related syscalls. Therefore, we need to distinguish the syscalls executed by the attackers from the rest of the syscalls introduced by our HoneyKube. To this end, we run an intrusion detection system (IDS) to detect suspicious behavior. Each time the IDS triggers an alert, we store the related syscalls for further investigation. We use a conservative strategy for this detection since collecting extra data from false alarms is preferable to missing important attack data. With this method, we improve the efficiency of the data analysis process while also reducing the number of collected systems trace files (SCAP). Although there is still a risk of missing out on some attack data, this method significantly improves the efficiency of the storage and analysis of the collected data, which would be otherwise unfeasible.

V. Implementation

While our design is platform-independent and can be run on any K8s compatible platform, we deploy HoneyKube on the Google Kubernetes Engine (GKE).

A. Monitoring Setup

Our monitoring infrastructure captures two main types of data: network traffic and system calls. The two mechanisms require different access levels within the system, and hence, we place them at separate observation points within the cluster. Figure 2 shows the architecture of these mechanisms in HoneyKube. Moreover, we configure the K8s infrastructure to collect additional information in log files, including K8s audit logs (i.e., logs from the control plane containing the chronological record of calls made to the K8s API server) and system logs from all computing units in the cluster.

Network Traffic Capture. We use a sidecar container running `tcpdump` to automatically collect all raw network traffic starting on pod creation. Due to the ephemeral nature of pods, PCAP files collected by `tcpdump` are written to persistent volumes inside pods, ensuring data is preserved when a pod is removed. While our `tcpdump` containers only passively observe traffic from other containers, advanced attackers may potentially attempt to gain access to these containers and the stored PCAP files. Thus, we periodically retrieve these files from our persistent volumes and store them locally.

System Calls Capture. We use Sysdig [38], which is a combination of several system-level monitoring tools, such as `strace` and `htop`, to capture system calls into system trace

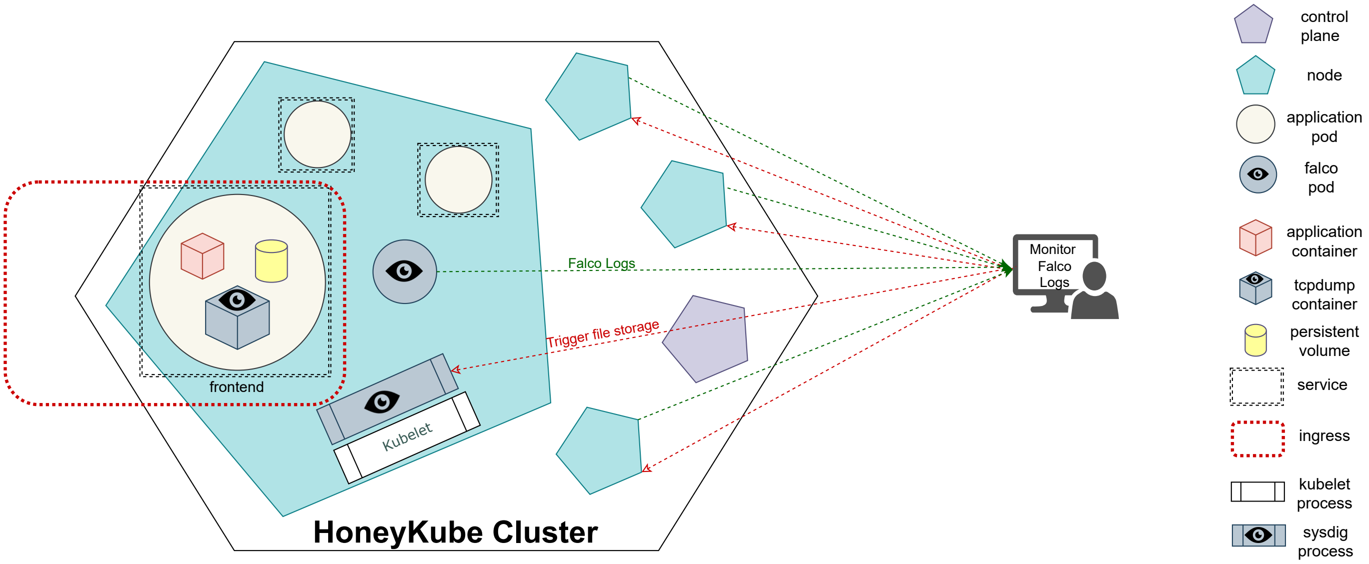

Fig. 2: **In-depth view of HoneyKube.** We have a cluster with four nodes and a control plane. To capture the network traffic, we run tcpdump on a sidecar container in each pod. We use persistent volumes in the pods to prevent the loss of the captured trace files. The Sysdig capture process runs on the node machines and uses circular file rotation. Falco (IDS) runs on a pod in each node and raises an alert on detecting suspicious behavior. The nodes use this alert to permanently store the captured trace files.

files (SCAP). Recall that we are mainly interested in capturing and filtering syscalls related to attacks in order to contain the size of the system trace files. Since syscall execution takes place at the kernel level, we set up the Sysdig process on each node. We configure this process to use a circular file storage mechanism where the oldest files are deleted as new ones are written, ensuring to store a fixed number (`n`) of files on the system. We run Falco [39], a cloud-native runtime security detection tool, to identify suspicious activities and raise alerts. Falco operates by running privileged containers on a separate pod on each node to access the host kernel. Whenever an alert is triggered, we store the syscall trace files permanently for the next 1000 seconds ($\sim$17 minutes) to prevent them from being deleted by the file rotation system. While Falco can actively notify us of alerts, such notifications could be intercepted by an attacker, making them aware of our honeypot. Hence, we passively poll Falco logs using `kubetail` [40].

B. Web Application

We base our decoy web application on the GCP microservices demo [37]. While the base application provides a realistic K8s microservices architecture, the application layout may be easily recognized as a fake webshop, and some of its microservices are not production-ready. Therefore, we redesign the application to make it more realistic. We migrate the product catalog to a MySQL database from a text file and populate it with new products. We also add functionalities for user login and registration, adding two additional microservices: the MySQL database and a Flask-based API service to access the database.

C. Vulnerabilities

We introduce security vulnerabilities in our honeypot based on the Azure Security Center's K8s threat matrix [30], [31]. This threat matrix is similar to the MITRE ATT&CK framework[2], consisting of tactics and techniques used by attackers in a K8s environment. We use this framework as a baseline for creating exploitable kill chains for attackers and map the added vulnerabilities to the adopted techniques. Injecting such vulnerabilities, we ensure that the attackers can exploit at least one path within the honeypot. At the same time, this does not limit their actions inside the cluster. The attackers can still take different steps to exploit other vulnerabilities specific to the K8s and microservices architecture.

Initial Access. We present the application as vulnerable to bait the attackers by printing errors and stack traces with software versions and database names on the user interface [41]. These bad coding practices make it evident that the application is vulnerable and ripe for exploitation. The information leaked via stack traces informs the attackers that the application uses version v1.11.5 of Golang, which is vulnerable to CRLF attacks (CVE-2019-9741 [42]). Moreover, we set up an SSH server in the frontend (UI) microservice with weak credentials (`admin:password123`) to enable the attackers to get access to the system (CWE-1391 [43]). We store these credentials in a publically available text file on the frontend of the web application, hashed with MD5. Through the *Vulnerable application* technique from the K8 threat matrix [30], [31], we offer attackers initial access to our cluster.

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

Execution. The next step in the kill chain is for attackers to execute arbitrary code within the cluster. A CRLF attack on the UI, developed with Golang v1.11.5, can further escalate into more malicious attacks such as Cross-Site Scripting (XSS) and page injection attacks(CWE-79 [44]). Additionally, we allow SQL injection attacks on our user database by not sanitizing user inputs before inserting them into the SQL queries (CWE-89 [45]). This enables the attackers to insert their commands into the SQL queries and access sensitive user information from other services within the cluster. And once the attackers gain initial access to the SSH server using the weak credentials, they can execute commands of their choice within the container. These vulnerabilities allow the *Application exploit* and *SSH server running inside container* techniques [30], [31] to be exploited by attackers.

Credential Access. After gaining shell access inside a container through SSH, the attackers can attempt to establish persistence, escalate privileges, or both. One possible way of doing so is by gaining access to credentials of K8s service accounts, which allows services running on a container to identify themselves and send (privileged) commands to the K8s API. We introduce a new service account on the frontend container since it is the point for initial entry and authorize it 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 authorizing the service account access to the secrets, we allow the attackers to exploit both *List Kubernetes secrets* and *Access container service account* techniques from the K8s threat matrix [30], [31].

Discovery and Lateral Movement. One of the prime differences between the architecture of existing honeypots and HONEYKUBE is that microservices use a distributed environment with multiple computing devices and containers. Hence, *Discovery* of other microservices and *Lateral Movement* between containers and nodes is an important aspect that attackers could exploit. The service account tokens retrieved from the K8s secrets consist of the access token of another vulnerable service account with increased permissions. These permissions allow the attacker to list the services and endpoints in the cluster while also authorizing them to update the services. Since the scope of the containers is limited to the libraries and packages required to run the application, we install command-line tools like `curl` and `wget` to some of the containers to facilitate the attackers in their attack. The attackers can then use these tools to download whatever they need to probe the network, increase their understanding of the environment, and move within the cluster. In short, we allow attackers to discover services using the *Access the Kubernetes API server* and *Network mapping* techniques and enable lateral movement through the *Container service account* and *Cluster internal networking* techniques.

VI. SECURITY MODEL

We employ security measures to limit the damage that attackers could cause if they succeed in getting complete control of our system. We restrict all unnecessary network traffic on the microservices; incoming traffic is allowed only on ports SSH, HTTP, and HTTPS; outgoing traffic is allowed only for HTTP and HTTPS. These policies prevent outgoing spam emails and reduce the chances of the system getting used for a DDoS attack. The containers in k8s have, by default, `root` privileges using which the attackers can escape the containers and gain access to the host machine. We prevent this escape by configuring the microservice containers, using Pod Security Context [46], to use a non-root and non-privileged user. For these configurations, we set `privileged: false`, `runAsNonRoot: true`, and `allowPrivilegeEscalation: false`. These settings prevent a process from gaining more privileges than its parent process and limit the damage the attackers can cause.

To ensure limited access to the control plane from application containers, we enable the Role-Based Access Control in GKE. We also configure the cluster to use shielded GKE nodes to prevent a known vulnerability in GKE [47]. Additionally, we limit the resources (CPU and RAM) available to each container to mitigate the execution of resource-intensive programs, such as crypto-mining or performing DDoS attacks from within the containers. Lastly, we add firewall policies to limit SSH port access on the node machines to only authorized IPs. We only allow the system we use for monitoring and data collection to access the node systems. These policies significantly reduce the possibilities for attackers to gain access to the nodes and/or gain root access to the system.

VII. EXPERIMENTAL SETUP

We conduct two experiments, an open one, where we expose HONEYKUBE to the Internet, and a controlled one, where we expose it to a set of recruited participants.

Environment. The HONEYKUBE cluster consists of 4 nodes and a control plane; the node machines use an E2 series high CPU machine with 4 vCPU and 4GB memory. We use Ubuntu OS in the nodes instead of the default Container-Optimized OS due to its support for Sysdig in GKE nodes.

Ethical Considerations. We reduce the risk of the misuse of the system by employing the security measures described in Section VI. The participants for the controlled experiment were volunteers who were briefed about the activity and had signed a consent form. We received approval from the ethics committee of our institution for both experiments.

Open Experiment. Recall that in k8s, Ingress exposes the services to the Internet using HTTP(S) protocols. For this experiment, we configure Ingress [48] to expose HONEYKUBE to the Internet using HTTPS protocol with a TLS certificate issued by the *Let's Encrypt* Certificate Authority. 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 on a separate IP. We add this IP to the *robots.txt* file to leak it to the attackers.

Controlled Experiment. We recruited two groups of participants for this experiment. One group included volunteers from our local CTF team. This group had no prior experience with Kubernetes or any microservices environments. The other group included volunteers from the red team of a medium-sized security operations company. The expertise levels of the members of this group varied from intermediate to high. The participants were given the instructions to attempt to infiltrate the honeypot and get access to the user's private information from the user database.

For this experiment, we restrict access to the HONEYKUBE to the participants' IP addresses. HONEYKUBE settings for the two experiments are identical except for one difference. In the controlled experiment, we directly expose the service (without Ingress) with a self-signed certificate, keeping the SSH service on the same IP address. With this experiment, we simulate targeted attacks on the system.

VIII. EXPERIMENTAL RESULT

A. Collected Datasets

We deployed HONEYKUBE on two separate clusters on GKE, one for each experimental setting. The open experiment was active for two weeks, while the controlled experiment was active for three weeks. We collected approximately 850 GB of data from the two experiments, consisting of system trace files, network trace files, and various log files.

System Trace Files. The system trace files (.scap files) capturing the system calls (with Sysdig) comprise most of the collected dataset (∼800 GB). The collection of these trace files was triggered whenever an alert was raised about suspicious activity in the system. Falco raised an alert for every SSH attempt and successful entry into the system. Since the open honeypot witnessed recurring brute-force attacks, Sysdig recorded a large number of trace files. We process these files to extract the executed system calls, their arguments, and execution timestamps.

Network Trace Files. The network trace files (.pcap files) were captured and stored in each pod. We collected approximately 8 GB of network trace files from the two experiments combined. These trace files recorded all inbound and outbound network traffic from every microservice.

K8s Audit Logs. Each request to the K8s API generates an audit event at every stage of its execution. These requests get recorded in the K8s audit logs. The collected log files amount to around 40 GB of data. The logs are stored in text files (*.json). These logs can help identify the effect of attackers' actions on the cluster as a whole.

Falco Logs. Falco stores all the events (alerts and otherwise) in log files. Since Falco is a security tool for cloud-native systems like K8s, its logs also contain details like pod names and container images where the alert was triggered. These details provide a better view of the attackers' movements and actions inside the cluster. The logs are stored in JSON files.

System Logs. All the default (debug) logs generated by the application containers and the audit logs generated by the nodes' OS are grouped under this category. We also collected the records of the login attempts (username and IP address) from the SSH server on the fronted service. All of these logs are stored in text files. We keep all these logs to provide context to the attackers' behavior, which can be identified by processing the collected data.

B. Fingerprintability

We used the Shodan [35] Honeyscore[3] tool to evaluate the fingerprintability of our HONEYKUBE. Honeyscore takes an IP address as input and fingerprints the corresponding device, computing the probability of the device being a honeypot. The output value ranges from 0.0 (real machine) to 1.0 (honeypot).

Although the details behind the computation of these Honeyscores are not publicly available, previous works [16] specify the criteria used for scoring honeypots (according to Shodan's developer): a) a large number of open network ports; b) the 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 most likely remains as a honeypot even after changing its configuration; e) a machine learning classification algorithm (not disclosed); f) known honeypots use the same configurations.

The Honeyscore tool assigned a value of 0.0 to our HONEYKUBE, which means this tool identified HONEYKUBE as a real system. The low score demonstrates the low fingerprintability of HONEYKUBE as a honeypot by this state-of-the-art reconnaissance tool. Since there are no known honeypots designed using the microservices architecture, it is safe to say that the Honeyscore tool has not yet been configured with fingerprintable markers to identify honeypots in this environment. Hence, this tool might not provide a reliable method to measure the fingerprintability of our honeypot. Nonetheless, 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.

Open Experiment. Within two weeks, the SSH server recorded approximately `11,500` login attempts from more than `200` distinct IP addresses, originating from `36` different countries. Only `12` out of approx. `11,500` successfully infiltrated the SSH server. Most of these immediately disconnected from the server after logging in, and the server continued to record more brute-force login attempts from those IP addresses. The timestamps and the haphazard nature of these attempts indicate automated brute-force attacks, most likely executed by bots scanning the internet for vulnerabilities. Table I lists the top five origins of these login attempts.

The SSH server logs recorded the usernames used for the login attempts. We observed `36` different usernames in these attempts. The most commonly observed usernames were `root`, `user`, `tech`, `demo`, and `telecomadmin`. By finding

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

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**

the vulnerable entry point to the cluster, the attackers got initial access to the honeypot.

A few of the attacks extended beyond just a successful entry. These attacks were able to infiltrate the cluster by brute-forcing the credentials of the SSH server. Once inside, these attacks attempted to execute system calls to meet their malicious requirements. The system calls captured from these attacks show a wide variety of behaviors. There was an attempt to mine Monero coins in one attack. A few others attempted to search for active crypto mining processes by using `grep` to find "Miner" or "miner" in the list of running processes. Whereas one read and deleted the system logs from the `/var/log` directory of the container. This is a classic example of *Defence Evasion* with an attacker covering their tracks [30], [31]. Based on the timestamps of these recorded system calls, it is evident that they were executed automatically, without manual intervention, or in the form of a script. Only one set of captured system calls consisted of delays between the commands, as shown in Figure 3. From the commands, it appears that there was an attempt to connect the container to a Dynamic Host Configuration Protocol (DHCP) server to achieve persistence in their connection to the cluster. We inferred this as possibly a manually executed attack.

Additionally, the cluster nodes recorded several brute-force login attempts as well. In Linux, all login attempts get logged under the `/var/log` directory by the `login` process. Successful and unsuccessful login attempts get stored in separate binary files (`wtmp` and `btmp` respectively). `btmp` files log the usernames, source IP addresses, and timestamps of all the failed login attempts. We observed that these attempts dated back to the creation of the cluster and not just the active period of the experiment. The attempts used `4000` distinct IP addresses, originating from `105` different countries.

Controlled Experiment. With this setting, we simulated a targeted attack scenario. Since the participants knew that HoneyKube was a honeypot, they approached it directly by searching for different vulnerabilities to exploit. The system logs retrieved from the containers show breach attempts on multiple interfaces. The two groups of participants used different techniques in their attacks. The CTF group had no prior experience with k8s. They manually searched the web platform for vulnerabilities and performed brute force attacks on the frontend service to gain access to the cluster. They mainly used `root` and `admin` as the username for the brute force attacks to get the credentials of the SSH server. While the group with the red team performed automated searches to find hidden web pages, automated SQL injection attacks on the user database via the API, and SSH brute-force attacks. They successfully retrieved data from the user database and used those credentials in their attempt to log into the SSH server as well. The infiltration tactics used by the participants are in line with the *Initial Access* and *Execution* tactics of the K8s threat matrix [30], [31].

```
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
```

Fig. 3: **The set of captured system calls from the possible manual attack**

We recorded nearly `1,500` system calls executed by the participants in this setting. Through these system calls, we observed that all the participants focused on identifying and understanding their environment, even though they used different approaches. The group of participants new to k8s downloaded generic tools like `nmap`[34] and `linPEAS`[49] to understand their environment. They manually searched the directory structure for clues and stumbled upon the service account tokens mounted on the container. Whereas the group familiar with k8s used k8s-specific tools like `kubectl` [50] and `peirates` [51], a K8s penetration tool, to assist in the discovery process. With `peirates`, these participants directly found the service account tokens. They then used the tokens to query the K8s API and get the list of services and secrets. The K8s secrets contained the credentials for the MySQL server. We noted that most participants attempted to connect to the database server with these credentials.

D. Discussion of Results

Our research shows that a honeypot developed using the microservices architecture is invariably different in design from one developed using monolithic architecture. As discussed in the previous section, most of the attacks recorded in the open setting were automated attacks. These were, most likely, executed by bots looking for vulnerable systems to add to their botnet or mine cryptocurrencies. None of the attacks recorded in the open setting specifically targeted the microservices or k8s environment. Neither were they designed to adapt to their environment. Hence, the attack behavior observed through the collected data in this setting did not vary from the monolithic systems.

We do not believe that the attack where the attacker read and deleted the system log files was successful since the container did not have the privilege to perform those actions on the node machine. And the system calls were recorded on the node level. So, they could not have been tampered with without getting access to the node machine. The attack attempt flagged as a possible manual attack 3 initiated a connection with a DHCP server. From the captured network traffic, we observed that it succeeded in establishing the connection. Unfortunately, this attack was performed just the day before the end of our open experiment. So, there is no way to know the attacker's plans for this connection.

The containers in a microservices architecture have a limited scope of capabilities. With the security measures employed for HONEYKUBE, the automated attacks were thwarted as they failed to gain root access. An example was the attempt to delete the system log files, which failed due to the lack of root access. Similarly, the attempts to mine cryptocurrencies failed due to the lack of root credentials and limited resources (CPU and memory) containers have at their disposal, as mining is a resource-intensive process.

The data collected from the controlled setting provides a much better outlook into the attackers' behavior when exploiting 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, were notably different. We observed from the system calls captured from the controlled experiment that participants using `linPEAS` switched to k8s-specific methods of discovering the environment after running this script. Since the k8s-specific penetration tools, such as `peirates`, are far more effective in obtaining credentials and enabling lateral movement within the cluster.

Lastly, we observed that the numerous components inside a microservices architecture are a challenge for security due to the number of interfaces available for the attacker to exploit. In the open setting, we saw this from the attacks on the cluster nodes. While in the controlled experiment, it became evident from the attempts to breach the system via different interfaces. In this research, we saw the differences in attackers' behaviors when attacking microservices architecture due to the larger attack surface. The extent of these differences still needs to be explored further. Repeating this experiment for a longer duration and analyzing the data collected, in addition to analyzing the data collected from this research, can help improve our understanding of these differences and improve the quality of the tools we develop to secure microservices architectures.

IX. LIMITATIONS AND FUTURE WORK

Here we discuss the limitations of our design and the different ways this research can move forward in the future.

Interaction Level. The quality of the collected data depends on the level of interaction given to the attackers. This interaction surface depends on the vulnerabilities in the given environment to 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. Despite the injected vulnerabilities, the attackers' movements within the system were a bit restricted, restraining the level of infiltration they could achieve. Adding more vulnerabilities along the entire cyber kill chain will allow higher mobility and a deeper infiltration by the attackers. Ultimately, this will facilitate the collection of higher-quality data about real-world attacks.

Monitoring and Data Collection. The monitoring mechanism we designed to capture the executed system calls for HONEYKUBE can be further improved. Our current design will only work if most of the attackers' actions inside the containers trigger alerts in Falco. If that is not the case, we might miss some activities inside the cluster. For example, in a scenario where the attackers enter the system and wait before starting to dig into it with benign commands like `ls` or `ps`, our monitoring system will not capture it. Secondly, the trigger mechanism used to capture system calls can be made more precise by avoiding collecting unnecessary system trace files. This can be achieved by sending triggers only to the node experiencing activities instead of all the nodes. Additionally, the practical 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 the storage of system calls can help improve the coverage of the collected data.

Experiment Duration. The types of observed 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., human attackers. Potentially, we can solve 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 to attract more genuine attacks and facilitate data collection from targeted or manual attacks.

Future Work. For future work, we plan to conduct a thorough analysis of the collected data to identify attack patterns and tools used by the attackers in exploiting microservices architecture. We also aim to repeat this experiment with different orchestrating and deployment platforms. The data collected from all of these experiments will be able to provide a better understanding of the attack surfaces of generic microservices architecture and not tied to specific platforms. The analysis of this data will assist in devising better security mechanisms for microservices-based applications.

X. RELATED WORK

With microservices and K8s taking the industry by storm, there has been plenty of research towards understanding their threat landscape and improving their security. In 2019, Tien et al. proposed KubAnomaly [52], which uses neural network approaches to create classification models that identify anomalies in the K8s environment. The resilience approach proposed by Baarzi et al. [53] focuses on preventing DDoS attacks by monitoring the resource usage of the microservices and quarantining them on a separate node on the identification of any abnormality. Works like the Cloud-native sandboxes [54] and Sandnet [55] use sandboxing techniques to record threat activities. The Cloud-native sandboxes focus on context-aware sandboxing techniques; Sandnet focuses on identifying compromised microservices and quarantining them in a sandboxed environment to monitor the threat in action. While Sandnet's goal is similar to ours with the HONEYKUBE, it had a few limitations. Firstly, the design can handle only one intrusion at a time. So, multiple adversaries within the same time frame are out of scope. Secondly, it only focuses on the network traffic behaviors of the attackers even after entering the cluster.

Although not much precedence exists in honeypots utilizing microservices architecture, there has been plenty of research towards innovative honeypots to catch up with the ever-evolving technology. As a result of the various cyberattacks on industries using Industrial Control Systems (ICS) like Stuxnet, Triton, and WannaCry, researchers designed varying honeypots emulating ICS systems [14]–[16], [56]–[58]. These honeypots aimed to improve the understanding of the threat landscape of ICS. One such high interaction honeypot [14], by Trend Micro Research, emulated a smart-factory solution for a fictitious company. The realistic factor of the solution was increased by adding the company backstory and employee contact details. During the seven months of activity, the honeypot recorded a large variety of cyberattacks like cryptocurrency mining as part of a botnet, multiple ransomware attacks, fingerprinting attacks with scanners, and many control systems attacks on the Programmable Logic Controllers (PLCs) in the system. While the data collected from this research gives a detailed view of the attacker's actions and motivations, this honeypot also highlights the underlying risks of using high-interaction honeypots. Overcoming the limitations of existing honeypot implementations for ICSs [56]–[58], HoneyPLC [16] developed high-interaction honeypots for PLCs within ICS. PLCs control critical systems like centrifuge machines in nuclear power plants. HoneyPLC proved its covertness by evaluating itself using Shodan, the state-of-the-art reconnaissance tool, and receiving a HoneyScore of 0.0. Some other innovative honeypots include HoneyMix [36], a software-defined networking (SDN) honeynet proposed to mitigate fingerprinting techniques, and "Honeypots-as-a-service" (HaaS) [13], which provides a scalable and flexible plug-and-play service for industrial honeypots.

XI. Conclusion

In this research, we introduced HoneyKube, a web honeypot designed using the microservices architecture. The main objective of the HoneyKube is to collect interaction data from real-world attacks. We used a real-world application as a baseline for HoneyKube to make it harder to fingerprint it as a honeypot. Furthermore, we increased the interaction surface of the honeypot by injecting vulnerabilities to improve the quality of the collected data. Through our experiments, we showed that the monitoring setup we devised for this architecture is effective in recording attackers' actions. We expect that our work and the collected data will foster new research to improve the security of microservices-based applications.

Acknowledgments

We would like to thank our reviewers for their valuable comments. This work has been supported by the INTERSECT project, Grant No. NWA 1160.18.301, funded by Netherlands Organisation for Scientific Research (NWO). The findings reported herein are the sole responsibility of the authors.

References

[1] ReportLinker, "Global cloud microservices market to reach $4.1 billion by 2030," Feb 2023. [Online]. Available: https://www.globenewswire.com/en/news-release/2023/02/01/2599585/0/en/Global-Cloud-Microservices-Market-to-Reach-4-1-Billion-by-2030.html

[2] A. Gluck, "Introducing domain-oriented microservice architecture," Jul 2020. [Online]. Available: https://eng.uber.com/microservice-architecture/

[3] K. Varshneya, "Understanding design of microservices architecture at netflix," Dec 2021. [Online]. Available: https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/

[4] E. Montalbano, "380k kubernetes api servers exposed to public internet," May 2022, [Date Accessed: 21 February 2022]. [Online]. Available: https://threatpost.com/380k-kubernetes-api-servers-exposed-to-public-internet/179679/

[5] T. Seals, "Argo cd security bug opens kubernetes cloud apps to attackers," Feb 2022, [Date Accessed: 21 February 2022]. [Online]. Available: https://threatpost.com/argo-cd-security-bug-kubernetes-cloud-apps/178239/

[6] G. Singer, "Threat alert: Kinsing malware attacks targeting container environments," Nov 2020, [Date Accessed: 21 February 2022]. [Online]. Available: https://blog.aquasec.com/threat-alert-kinsing-malware-container-vulnerability

[7] C. Cimpanu, "Vast majority of cyber-attacks on cloud servers aim to mine cryptocurrency," Sep 2020, [Date Accessed: 21 February 2022]. [Online]. Available: https://www.zdnet.com/article/vast-majority-of-cyber-attacks-on-cloud-servers-aim-to-mine-cryptocurrency/

[8] L. Vaas, "Windows container malware targets kubernetes clusters," [Date Accessed: 21 February 2022]. [Online]. Available: https://threatpost.com/windows-containers-malware-targets-kubernetes/166692/

[9] T. Seals, "Poorly secured docker image comes under rapid attack," April 2020, [Date Accessed: 21 February 2022]. [Online]. Available: https://threatpost.com/poorly-secured-docker-image-rapid-attack/154874/

[10] A. El Khairi, M. Caselli, C. Knierim, A. Peter, and A. Continella, "Contextualizing system calls in containers for anomaly-based intrusion detection," in *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, November 2022.

[11] Y. Lin, O. Tunde-Onadele, and X. Gu, "Cdl: Classified distributed learning for detecting security attacks in containerized applications," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

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

[13] J. H. Jafarian and A. Niakanlahiji, "Delivering honeypots as a service," *Proceedings of the Annual Hawaii International Conference on System Sciences*, Jan 2020.

[14] S. Hilt, F. Maggi, C. Perine, L. Remorin, M. Rösler, and R. Vosseler, "Caught in the act: Running a realistic factory honeypot to capture real threats," Jan 2020. [Online]. Available: https://documents.trendmicro.com/assets/white_papers/wp-caught-in-the-act-running-a-realistic-factory-honeypot-to-capture-real-threats.pdf

[15] P. Ferretti, M. Pogliani, and S. Zanero, "Characterizing background noise in ics traffic through a set of low interaction honeypots," *Proceedings of the ACM Workshop on Cyber-Physical Systems Security & Privacy*, Nov 2019.

[16] E. López-Morales, C. Rubio-Medrano, A. Doupé, Y. Shoshitaishvili, R. Wang, T. Bao, and G.-J. Ahn, "Honeyplc: A next-generation honeypot for industrial control systems," *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Nov 2020.

[17] S. Dowling, M. Schukat, and E. Barrett, "New framework for adaptive and agile honeypots," *ETRI Journal*, vol. 42, no. 6, p. 965–975, July 2020.

[18] E. Vasilomanolakis, S. Karuppayah, P. Kikiras, and M. Mühlhäuser, "A honeypot-driven cyber incident monitor: lessons learned and steps ahead," *Proceedings of the 8th International Conference on Security of Information and Networks*, Sept 2015.

[19] J. Safarik, M. Voznak, F. Rezac, P. Partila, and K. Tomala, "Automatic analysis of attack data from distributed honeypot network," *Proceedings of Society of Photo-optical Instrumentation Engineers (SPIE)*, May 2013.

[20] M. Nawrocki, M. Wählisch, T. C. Schmidt, C. Keil, and J. Schönfelder, “A survey on honeypot software and data analysis,” *ArXiv*, vol. abs/1608.06249, Aug 2016.

[21] S. Khattab, R. Melhem, D. Mosse, and T. Znati, “Honeypot back-propagation for mitigating spoofing distributed denial-of-service attacks,” *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, July 2006.

[22] S. Eftimie and C. Răcuciu, “Honeypot system based on software containers,” *Scientific Bulletin of Naval Academy*, vol. 19, no. 2, p. 415–418, 2016.

[23] V.-H. Pham and M. Dacier, “Honeypot trace forensics: The observation viewpoint matters,” *Future Generation Computer Systems*, vol. 27, no. 5, p. 539–546, June 2011.

[24] D. Fraunholz, M. Zimmermann, and H. D. Schotten, “An adaptive honeypot configuration, deployment and maintenance strategy,” *Proceedings of the International Conference on Advanced Communication Technology (ICACT)*, Feb 2017.

[25] A. Pauna and I. Bica, “Rassh - reinforced adaptive ssh honeypot,” *Proceedings of the International Conference on Communications (COMM)*, May 2014.

[26] A. Gluck, “Introducing domain-oriented microservice architecture,” Sep 2020. [Online]. Available: https://eng.uber.com/microservice-architecture/

[27] “Production-grade container orchestration,” [Date Accessed: 21 February 2022]. [Online]. Available: https://kubernetes.io/

[28] “Kubernetes service,” [Date Accessed: 23 Feb 2023]. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/

[29] “Kubernetes ingress,” [Date Accessed: 21 February 2022]. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/

[30] “Threat matrix for kubernetes,” Nov 2021. [Online]. Available: https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/

[31] “Secure containerized environments with updated threat matrix for kubernetes,” Mar 2021. [Online]. Available: https://www.microsoft.com/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/

[32] T. van Ede, R. Bortolameotti, A. Continella, J. Ren, D. J. Dubois, M. Lindorfer, D. Choffnes, M. van Steen, and A. Peter, “Flowprint: Semi-supervised mobile-app fingerprinting on encrypted network traffic,” *Proceedings of the Network and Distributed System Security Symposium*, Feb 2020.

[33] T. van Ede, H. Aghakhani, N. Spahn, R. Bortolameotti, M. Cova, A. Continella, M. van Steen, A. Peter, C. Kruegel, and G. Vigna, “Deepcase: Semi-supervised contextual analysis of security events,” *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2022.

[34] G. Lyon, *Nmap network scanning: Official NMAP project guide to network discovery and security scanning*. Insecure. Com LLC, 2008.

[35] J. Matherly, *Complete guide to Shodan (2016-02-25)*. Shodan, LLC, 2016.

[36] W. Han, Z. Zhao, A. Doupé, and G.-J. Ahn, “Honeymix : Toward sdn-based intelligent honeynet,” *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, March 2016.

[37] GoogleCloudPlatform, “Googlecloudplatform/microservices-demo: Sample cloud-native application with 10 microservices showcasing kubernetes, istio, grpc and opencensus.” [Date Accessed: 21 February 2022]. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[42] Cvedetails.com, “Vulnerability details : Cve-2019-9741,” [Date Accessed: 21 February 2022]. [Online]. Available: https://www.cvedetails.com/cve/CVE-2019-9741/

[38] Draios, “Draios/sysdig: Linux system exploration and troubleshooting tool with first class support for containers,” [Date Accessed: 21 February 2022]. [Online]. Available: https://github.com/draios/sysdig

[39] Falcosecurity, “Falcosecurity/falco: Cloud native runtime security,” [Date Accessed: 21 February 2022]. [Online]. Available: https://github.com/falcosecurity/falco

[40] J. Haleby, “Johanhaleby/kubetail: Bash script to tail kubernetes logs from multiple pods at the same time,” [Date Accessed: 21 February 2022]. [Online]. Available: https://github.com/johanhaleby/kubetail

[41] “Information disclosure vulnerabilities,” [Date Accessed: 21 February 2022]. [Online]. Available: https://portswigger.net/web-security/information-disclosure

[43] “Cwe-1391: Use of weak credentials,” [Date Accessed: 29 March 2023]. [Online]. Available: https://cwe.mitre.org/data/definitions/1391.html

[44] “Cwe-79: Improper neutralization of input during web page generation (’cross-site scripting’),” [Date Accessed: 29 March 2023]. [Online]. Available: https://cwe.mitre.org/data/definitions/79.html

[45] “Cwe-89: Improper neutralization of special elements used in an sql command (’sql injection’),” [Date Accessed: 29 March 2023]. [Online]. Available: https://cwe.mitre.org/data/definitions/89.html

[46] “Configure a security context for a pod or container,” [Date Accessed: 21 February 2022]. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

[47] M. Wickenden, “Hacking kubelet on google kubernetes engine,” Nov 2018. [Online]. Available: https://www.4armed.com/blog/hacking-kubelet-on-gke/

[48] “Ingress for external http(s) load balancing — kubernetes engine documentation — google cloud,” [Date Accessed: 21 February 2022]. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/ingress-xlb

[49] C. Polop, “Peass-ng/linpeas at master · carlospolop/peass-ng,” [Date Accessed: 21 February 2022]. [Online]. Available: https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS

[50] “Kubectl,” [Date Accessed: 21 February 2022]. [Online]. Available: https://kubernetes.io/docs/reference/kubectl/

[51] Inguardians, “Inguardians/peirates: Peirates - kubernetes penetration testing tool,” [Date Accessed: 21 February 2022]. [Online]. Available: https://github.com/inguardians/peirates

[52] C. Tien, T. Huang, C. Tien, T. Huang, and S. Kuo, “Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches,” *Journal of Engineering Reports*, vol. 1, no. 5, Dec 2019.

[53] A. F. Baarzi, G. Kesidis, D. Fleck, and A. Stavrou, “Microservices made attack-resilient using unsupervised service fissioning,” *Proceedings of the European workshop on Systems Security (EuroSec)*, p. 31–36, April 2020.

[54] Z. Xu and T. Luo, “Cloud-native sandboxes for microservices: Understanding new threats and attacks,” *Blackhat Europe*, 2019.

[55] A. Osman, P. Bruckner, H. Salah, F. H. Fitzek, T. Strufe, and M. Fischer, “Sandnet: Towards high quality of deception in container-based microservice architectures,” *Proceedings of the IEEE International Conference on Communications (ICC)*, May 2019.

[56] D. I. Buza, F. Juhász, G. Miru, M. Félegyházi, and T. Holczer, “Cryplh: Protecting smart energy systems from targeted attacks with a plc honeypot,” *Lecture Notes in Computer Science*, vol. 8448, p. 181–192, Jan 2014.

[57] S. Hilt, “Sjhilt/gaspot: Gaspot released at blackhat,” [Date Accessed: 21 February 2022]. [Online]. Available: https://github.com/sjhilt/GasPot

[58] A. Jicha, M. Patton, and H. Chen, “Scada honeypots: An in-depth analysis of conpot,” *Proceedings of the IEEE Conference on Intelligence and Security Informatics (ISI)*, Nov 2016.