On Detecting Malicious Code Injection by Monitoring Multi-Level Container Activities

Md. Olid Hasan Bhuiyan[1], Souvik Das[1], Shafayat Hossain Majumder[2], Suryadipta Majumdar[2] and Md. Shohrab Hossain[1]

[1]*Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Bangladesh*

[2]*Concordia Institute for Information Systems Engineering, Concordia University, Canada*

Keywords: Container Security, Malicious Code Injection, Kubernetes, Multi-Level Monitoring.

Abstract: In recent years, cloud-native applications have been widely hosted and managed in containerized environments due to their unique benefits, such as being lightweight, portable, and cost-efficient. Their growing popularity makes them a common subject of cyberthreats, as evidenced by recent attacks. Many of those attacks take place due to malicious code injection to breach systems and steal sensitive data from a containerized environment. However, existing solutions fail to classify malicious code injection attacks that impact multiple levels (e.g., application and orchestrator). In this paper, we fill in this gap and propose a multi-level monitoring-based approach where we monitor container activities at both the system call level and the container orchestrator (e.g., Kubernetes) level. Specifically, our approach can distinguish between the expected and unexpected behavior of a container from various system call characteristics (e.g., sequence, frequency, etc.) along with the activities through event logs at the orchestrator level to detect malicious code injection attacks. We implement our approach for Kubernetes, a major container orchestrator, and evaluate it against various attack paths outlined by the Cloud Native Computing Foundation (CNCF), an open-source foundation for cloud native computing.

1 INTRODUCTION

The use of containers has substantially increased over the past several years along with their popularity. According to RedHat's 2023 report (Redhat, 2023), 97% of IT leaders report seeing business benefits of cloud containerization technologies, such as Kubernetes, Docker, etc. As container technology proliferates, so does its susceptibility to security threats (Song et al., 2023; Lee et al., 2023a; Jang et al., 2022). Recent discovery of active campaigns deploying backdoors/malware in Kubernetes clusters have impacted over 350 companies, including Fortune 500 firms, which highlights significant risks of this technology (Aqua, 2023). Docker containers, a frequent target for code injection, e.g., crypto mining exploits (Cuen, 2018), and compromises in Docker Hub (DockerHub, 2022) further emphasize the gravity of security breaches in container technology.

Malicious code injection poses a severe threat to containerized environments (VS et al., 2023; Wong et al., 2023), enabling attackers to compromise data privacy, disrupt operations, or exploit containers for additional attacks. Attackers exploit application code vulnerabilities to introduce malicious code, exemplified by our simulated research scenarios in section 3.3. To make things worse, the dynamic and elusive nature of containers complicates detecting and preventing such injections, impeding timely detection, especially when attackers gain access to the host system. Moreover, the diverse access routes provided by containers challenge conventional detection systems, hindering swift response.

The existing works (e.g., (Abed et al., 2015; Castanhel et al., 2021; Cavalcanti et al., 2021; Brown et al., 2022; Lee et al., 2023b)) on detection of malicious code injection faces the following challenges:

- Containers offer a broad attack surface, presenting ample opportunities for malicious code injection at various levels (e.g., application, orchestrator). Existing works lack comprehensive monitoring of these container levels (Fig. 1).
- To counteract malicious code injection, leverag-

Bhuiyan, M., Das, S., Majumder, S., Majumdar, S. and Hossain, M.
On Detecting Malicious Code Injection by Monitoring Multi-Level Container Activities.
DOI: 10.5220/0012509400003711

In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)*, pages 15-26
ISBN: 978-989-758-701-6; ISSN: 2184-5042

ing system call analysis proves to be promising(Abed et al., 2015; Castanhel et al., 2021; Cavalcanti et al., 2021). Current solutions use a generic syscall-based approach to tackle myriads of attack pathways. However, because each attack is unique, each of these require a specialized detection mechanism. Thus, existing solutions fall short (in trying to effectively counter all the attacks with one generic syscall-based approach).

- During malicious code injection, not all system calls are equally important for analyzing anomalous behavior, and many of these can be identified harmless (Castanhel et al., 2021). If we filter out these unnecessary system calls, attack detection will be more effective.

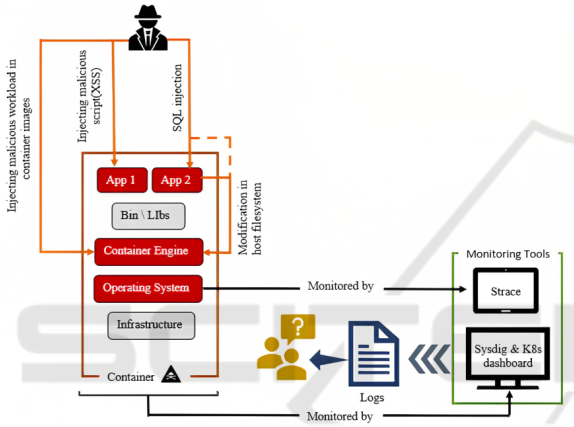


Figure 1: Proposed method to monitor different levels of container through different monitoring tools.

In this paper, we propose a synergistic blend of varying methods for monitoring containers in a multi-level fashion, considering both container level logs (syscalls), as well as orchestrator-level event logs to counteract the multi-path nature of these attacks, as seen in Figure 1. Focusing on various container levels, our work provides multiple detection mechanisms since malicious code injection can take place at different levels of container for distinct attack pathways. We propose deploying a specialized container within the Kubernetes environment dedicated to implementing our approach for detecting malicious code injections. This complements existing ML-based approaches by adding new methods (like using frequency and orchestrator-level monitoring) to detect malicious code injection, enhancing the capability to identify attacks occurring along various paths. Leveraging attack paths outlined by the Cloud Native Computing Foundation (CNCF) Finance User Group (Lancini, 2020), we employ scenarios relevant to container deployment with Kubernetes orchestration, encompassing numerous potential routes for malicious code injection. Our main contributions are:

- We are the first to propose a multi-level (e.g., application, operating system and orchestrator) monitoring approach which enables more robust detection of malicious code injection where the attacks can only be identified by checking container activities at multiple levels.
- In our analysis of malicious code injection attacks on container behavior, specifically focusing on key stages outlined in the CNCF framework, we delve into critical factors like syscall frequency and sequence. This comprehensive examination encompasses various routes for a broad spectrum of malicious code injection attacks. To enhance the efficiency of our solution, we categorize a key group of system calls that exhibit substantial changes during an attack, utilizing frequency as a filtering mechanism to exclude irrelevant system calls and better detect malicious activity.
- We implement our solution for Docker and Kubernetes (the most popular choices for containerization) and evaluate its effectiveness using the proposed attack paths in CNCF Finance User Group (CNCF, 2022) in comparison to existing works (Abed et al., 2015) and (Castanhel et al., 2021). We achieved a F1 score of 98.2%(Fig. 6b) and a recall score of 99%(Fig. 6a).

The rest of the paper is organized as follows. In Section 2, we discuss some existing works on container security. Section 3 presents the details of our proposed solution. Section 4 contains our implementation setup and dataset handling. In Section 5, we discuss the outcome of our experiment. Then in Section 6 we discuss the validity and reliability of our study. Finally, we conclude the paper in Section 7.

2 BACKGROUND AND LITERATURE REVIEW

Background. *Container:* Containers enable multiple instances of a program to run on a single host OS by sharing its kernel, providing separated user space, file system, and network stack, resulting in lightweight and efficient virtualization using portable images.

OS Level Virtualization: Multiple microservices may be deployed on a single server thanks to virtualization, which also offers cost savings, resource efficiency, and scalability. Due to its benefits over VMs, such as lightweight, fast, easy to deploy, effective resource usage, and support for version control, container-based virtualization stands out as the best choice for microservices (Sultan et al., 2019).

System call: User space applications can seek access to OS services by making system calls. The "kernel namespace isolation" feature of container systems enables containers to have its own environment, assuring isolation and security. The Seccomp (Secure Computing Mode) (Docker, 2022) feature of container runtimes limits system calls to combat attacks that use system call vulnerabilities.

Existing works. Brown *et al.* (Brown et al., 2022) use system calls in a performant model such as a random forest which allows fast, accurate classification in *certain* situations. Abed *et al.* (Abed et al., 2015) advocate a host-based intrusion detection solution for Linux containerized systems. Cavalcanti, Inácio, and Freire (Cavalcanti et al., 2021) propose a container-level anomaly-based intrusion detection system for multi-tenant applications using machine learning algorithms. Lin *et al.* (Lin et al., 2018) curate a dataset of attacks targeting the container platform, categorizing them using a two-dimensional attack taxonomy and assessing the security of Linux container technology with common attacks. Chelladhurai *et al.* (Chelladhurai et al., 2016) address Docker container security challenges, while Tunde *et al.* (Tunde-Onadele et al., 2019) propose a detection mechanism based on the frequency of selective system calls. These works cover various attack types, including SQL injection, system user privilege escalation, authentication circumvention etc. While they employ a common solution using syscall sequences, our work introduces attack-specific detection methods.

Yarygina et al. (Yarygina and Otterstad, 2018) propose a cost-conscious intrusion response system for microservices utilizing a game-theoretic method to respond to network threats in real time. Thanh Bui's (Bui, 2015) research focuses on Docker's internal security and its interactions with Linux kernel security features. Souppaya *et al.* (Souppaya et al., 2017) discuss security issues related to container usage and provide suggestions for resolution. Sarkale, Rad, and Lee (Sarkale et al., 2017) suggest a security layer using the Most Privileged Container (MPC). Lee *et al.* (Lee et al., 2023b) analyzes attack scenarios based on attack cases by malicious code, and surveys and analyzes attack techniques used in attack cases. While these works offer insights into container security measures, none of them implements and evaluates their methodologies using performance metrics or specifically addresses attack signatures, making it challenging to confirm their effectiveness, unlike our approach. Castanhel *et al.* (Castanhel et al., 2021) and Hofmeyr and Somayaji (Hofmeyr et al., 1998) work on detecting anomaly using sequence of system calls and different machine learning algorithms. They do not work on any particular attacks that can finally lead to malicious code injection.

While there are works unrelated to containerized environments, such as (Kuang and Zulkernine, 2008)'s intrusion-tolerant approach for Network Intrusion Detection Systems (NIDS), and DIGLOSSIA (Son et al., 2013) for code injection threat detection, their solutions can be adapted for container use. Kuang and Zulkernine focus on making NIDS intrusion-tolerant with independent components, while Son et al.'s tool, although effective, is implemented in a non-containerized environment, seperating it from our container-centric approach.

3 METHODOLOGY

This section describes our proposed method to detect malicious code injection by monitoring multi-level container activities.

3.1 Approach Overview

As illustrated in Figure 2, we streamline the entire process of monitoring container activities into two stages: pre-training, and runtime detection and monitoring. The pre-training phase involves the simulation of both benign and malicious behavior through diverse attack scenarios. By scrutinizing changes in the generated logs from both the container and orchestrator, we obtain valuable pre-training knowledge. This knowledge helps us in the later stage to perform efficient detection of code injection attacks. Throughout both of the stages, we leverage the following three detection mechanisms: log file analysis, syscall frequency assessment, and syscall sequence evaluation.

In the log file-based pre-training (Method 1), we identify crucial events (1a) and monitor their significance (1b). For frequency-based pre-training (Method 2), we collect system call frequencies (2a), identify impactful calls (2b), and establish standardized frequencies (2c). Finally, in sequence-based pre-training (Method 3), we gather syscall sequences (3a), filter unnecessary parameters (3b), and create standardized sequences using the Bag of System Calls (BoSC) technique (3c). Individual pre-training knowledge is extracted from each of these three methods.

During detection and monitoring phase, a monitor is deployed as a separate container, incorporating this pre-training knowledge alongside the current state of the cluster. Here, for log file based detection (Method 1), we event profiling to detect RBAC attacks (1c). For frequency based detection (Method

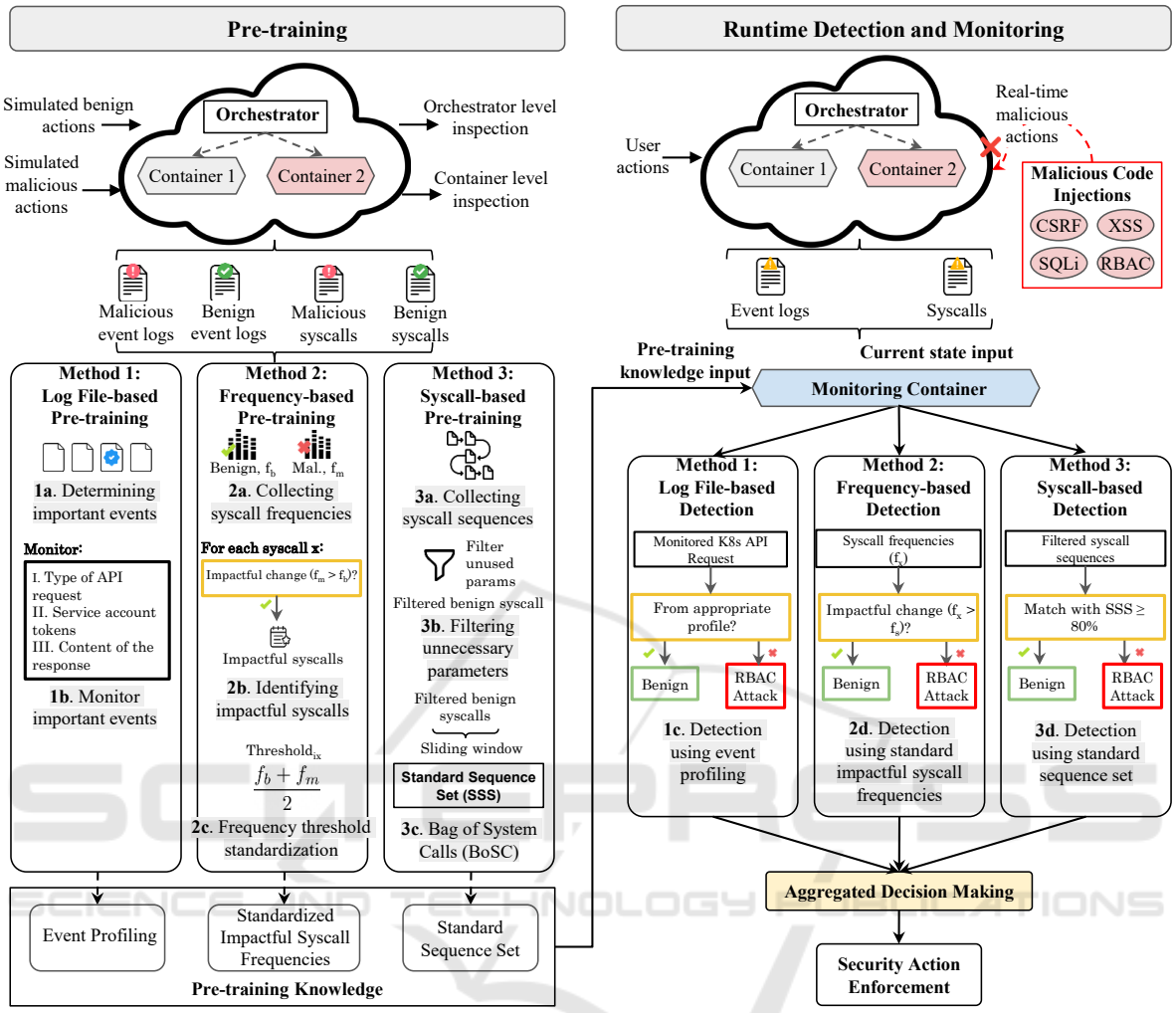


Figure 2: Overview of our proposed approach.

2), we employ impactful syscall frequencies for XSS attacks (2d). And lastly for sequence-based detection (Method 3), we use the Standard Sequence Set (SSS) to detect CSRF and SQLi attacks (3d). As each detection method specializes in a distinct class of attacks, their outcomes are aggregated to conclusively detect the presence or absence of an attack. This entire approach aids analysts in determining the necessary course of action by detecting code injection attacks from multiple facets. The subsequent sections elaborate on various decisions made as foundation of the solution.

3.2 Modeling Benign Behavior

Our approach in detecting malicious code injection at first involves figuring out the usual frequency of benign activities in order to create benign profiles. This is accomplished by building a comprehensive training set that combines malicious and benign data. This comprehensive dataset serves as the foundation for model training and identifying trends in system calls linked to both benign and malicious activity.

Crucial to our approach is the determination of the top k influential system calls (2a). Based on the frequency transitions between benign and malicious activity, we evaluate these system calls (2b). Because these transitions are dynamic, we may identify important system calls that show notable differences in their frequency patterns when they move from benign to malicious states.

We take advantage of the average frequency seen in both malicious and benign components to set effective thresholds for each system call. We can compute upper and lower bounds for the threshold values due to this dual consideration. As shown by 2(c) in Figure

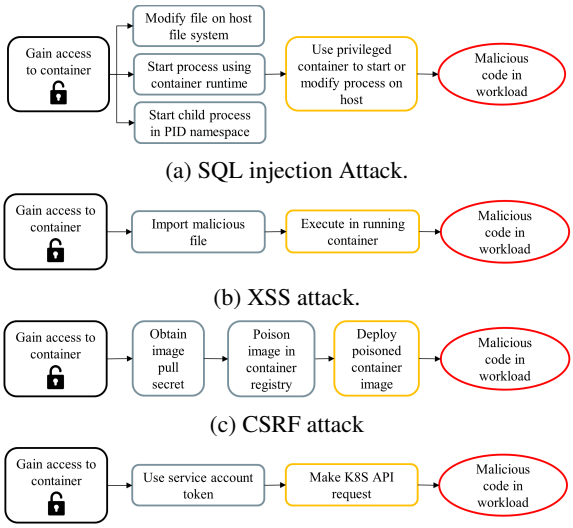
2, we improve the accuracy of our detection methods by including these upper and lower constraints in our analysis.

In Step 3(c) of Fig. 2, we conduct benign profile editing to capture both standard and malicious sequences of system calls. The standard sequence represents the system calls when a user visits a normal profile, while the malicious sequence portrays the system calls when an attacker injects something malicious into their profile, and a normal user subsequently gets attacked. Employing the Bag of System Calls (BoSC) and sliding window techniques (Abed et al., 2015), we extract sliding windows from the complete list of system calls, ranging in length from two to ten. The resulting frequency list, exemplified as $[0,1,1,1,0,0,0,0,1,2,2,0,0,0,0,0,1,0,0,1]$ for 20 system calls, is generated by listing the frequency of each system call in order. A new sequence is created by sliding one window of system calls, and if not already produced, its frequency list is added to our unique bag of sequences, disregarding duplicates.

Another approach of our methodology is to include data of important events happening in container through Kubernetes dashboard. The log of all API requests made to the master node is tracked by this Kubernetes dashboard. Kubernetes-entrypoint makes direct queries to the Kubernetes API. Consequently, all information related to an API request, including the content of the response and the service account token used for the request, is recorded in the Kubernetes log file (shown in 1(a) and 1(b) of Fig. 2). Later on, these recorded contents are utilized to determine whether or not a service account made an unauthorized API request.

Finally, we record the names of all system calls made by a user who does not have root access to a container. We consider this to be a list of common system calls performed by non-root users. We carefully follow the procedures described in this section in order to create a benchmark. Every system call that has been detected is examined for its frequency of occurrence and its sequence under benign conditions. Through this procedure, a baseline understanding of the normal behavior displayed by non-root users in the container environment may be created. Now that we have this baseline, we can turn our attention to proactively detecting possibly malicious activity. We can now identify irregularities from the predetermined benchmark by analyzing system calls made inside the container. We then utilize this benchmark to identify the injection of malicious code into a container.

For the additional assessment that simulates benign behavior, we improve the frequency-based



(a) SQL injection Attack.

(b) XSS attack.

(c) CSRF attack

(d) RBAC attack.

Figure 3: Attack paths of SQLi, XSS, CSRF and RBAC.

analysis by incorporating event-number and event-timestamp information for every system call. The goal of this integration is to distinguish between normal high-workload situations and possible security breaches. During this assessment, we meticulously monitor the disparity in timestamps between successive system call events, as well as the difference in event numbers. Specifically, if the event-number difference between two consecutive events is notably high, yet the timestamp difference remains significantly low, we interpret this pattern as indicative of a potential security attack. With this method, we can quickly spot anomalous trends in system call activity. We are able to identify and address possible security issues by concentrating on scenarios in which there is an anomalous surge in the difference between the event numbers while keeping the timestamps reasonably small.

3.3 Simulating Attacks

In this section, we are going to simulate four different attacks which finally result in malicious code injection in container. Each of these attacks represents different attack paths presented in Fig. 3.

Attack Overview. We identify four distinct attack paths provided by CNCF that ultimately lead to malicious code injection. These paths include SQLi (Fig. 3a), XSS (Fig. 3b), CSRF (Fig. 3c), and RBAC (Fig. 3d). These attacks are simulated using SEED-LABS 2.0 (SeedLab, 2022), with the exception of RBAC, which follows a custom implementation.

For these alternative attack pathways, we simulate various attacks using a social networking plat-

form from SEED-LABS 2.0 for XSS and CSRF, and a company website for SQLi. The social platform allows users to edit profiles, view others' profiles, and send friend requests. Meanwhile, the company website, deployed with two containers, enables employees to view and modify certain profile fields.

CSRF Attack. This attack is done using malicious code injection that follows the path shown in Fig. 3c. The attack scenario is combining a malicious container that contains elements that are malicious, like malicious image, with a benign social networking website. Because it makes it possible to take advantage of vulnerabilities in the benign container, this integration is essential to the attack's success.

When a person visits the compromised website, the malicious code injection is especially designed to alter their profile without their knowledge or agreement. The attack mechanism involves a concealed POST request directed towards the benign container encapsulated within the image of the malicious container. Under the pretense of legitimate traffic, this covert communication enables the malicious code that has been inserted to interact with and modify the user's profile on the benign website. The subtle nature of Cross-Site Request Forgery (CSRF) attacks is illustrated by the attacker's ability to successfully carry out unauthorized modifications to the user's profile by taking advantage of the trust that has been formed between the user and the compromised container (seed-labs 2.0, 2022a).

XSS Attack. To carry out this attack, the attacker uses malicious code injection by following the path that is marked out in Fig.3b. By placing a malicious script on the attacker's profile, the attack's method entails infecting a benign website and gaining control over the profiles of other users. An essential component of this attack is the intentional alteration of the targeted user's profile without their awareness, which occurs when the user visits the attacker's profile page. By carefully modifying their own profile, the attacker inserts malicious code into the material linked to their profile page on the trustworthy website.

When other users interact with the compromised profile, the injected code acts as a vector to spread the attack. By use of this deceptive distribution of malicious content, the attacker manages to subtly alter other users' profiles, so extending the attack's reach throughout the benign website(seed-labs 2.0, 2022b).

SQLi Attack. Executed through malicious code injection as depicted in Fig.3a, this attack method hinges on exploiting vulnerabilities via SQL injection (SQLi). To manipulate the database tables, the attacker strategically utilizes the login page to insert a malicious SQL query. A crucial prerequisite for the effectiveness of this attack is the attacker's prior familiarity with the database table structure, allowing them to craft a query that exploits potential weaknesses. Through the login page, the attacker gains access to the system and uses the input fields to insert a carefully crafted SQL query. If successful, this inserted query modifies the underlying database tables, which may allow for illegal access, the retrieval of data, or even the change of data(seed-labs 2.0, 2022c).

RBAC Attack. Executed through malicious code injection along the path illustrated in Fig. 3d, this attack method leverages a service account endowed with the privilege, or role, of generating containers within the kube-system namespace. This simulation emulates a Role-Based Access Control (RBAC) attack against Kubernetes, highlighting vulnerabilities that may be exploited by adversaries. Once a container is successfully created in the master node under the disguise of the privileged service account, the attacker gains the opportunity to inject a malicious workload into the container. This specific attack presents a serious security concern since the malicious container has access to every service account in the kube-system namespace. With that much access, the malicious container might potentially jeopardize the privacy of critical data by stealing credentials from different service accounts in the kube-system namespace. This unauthorized access puts the entire security posture of the Kubernetes system at serious risk by enabling the attacker to stealthily gather and exploit vital credentials (Gerzi, 2022).

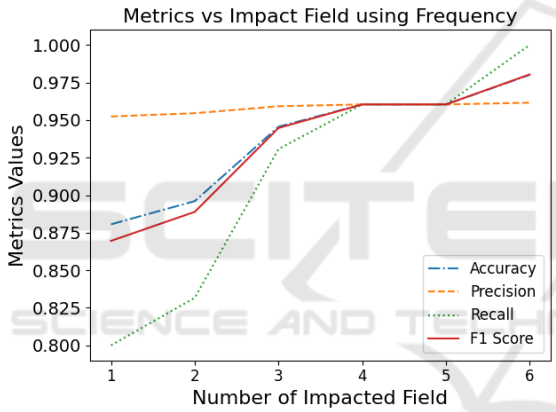
3.4 Differentiating Malicious Code Injection from Benign Activities

We set a baseline for system call frequency during usual activities, including profile visits, in the initial phase of our study. The subsequent detection of anomalies process used this baseline as a point of reference. We have now purposefully included Cross-Site Scripting (XSS) attacks of varied intensities into user profiles in order to thoroughly assess our system's resistance to them. The number of modified fields is changed from one to six, which changed the attack's severity. Our detection technique identifies possible threats by methodically comparing frequency samples obtained during these intentional attacks to our defined normal reference.

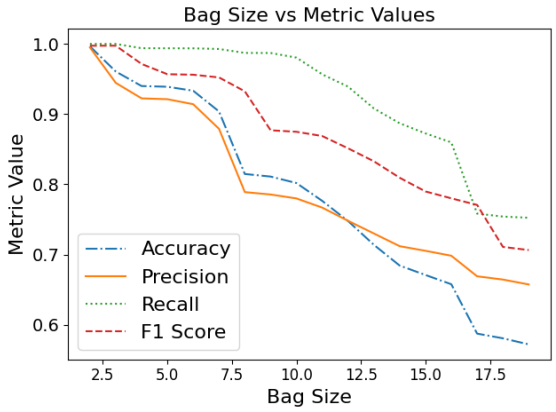
Specifically, the activity is flagged as possible XSS attack if more than 82% of the chosen system calls exceed our predetermined frequency threshold. As explained in Step 2(d) of Fig. 2, the crucial threshold of 82% is determined by a rigorous trial-and-error procedure aimed at attaining maximum recall score.

Table 1: Definition of sets of system calls for Fig. 5.

Values along X-axis for all subfigures in fig 5						
Set	Modified Field=1	Modified Field=2	Modified Field=3	Modified Field=4	Modified Field=5	Modified Field=6
S1	write	pwrite	fdatasync	pread	write	fdatasync
S2	S1+fdatasync	S1+write	S1+write	S1+fdatasync	S1+fdatasync	S1+write
S3	S2+pwrite	S2+fdatasync	S2+pwrite	S2+write	S2+pwrite	S2+pwrite
S4	S3+pread	S3+fsync	S3+fsync	S3+pwrite	S3+fsync	S3+do_submit
S5	S4+fsync	S4+io_submit	S4+io_submit	S4+fsync	S4+io_submit	S4+fsync
S6	S5+io_submit	S5+sched_yield	S5+sched_yield	S5+io_submit	S5+sched_yield	S5+io_submit
S7	S6+sched_yield	S6+futex	S6+futex	S6+sched_yield	S6+nanosleep	S6+sched_yield
S8	S7+nanosleep	S7+nanosleep	S7+nanosleep	S7+nanosleep	S7+futex	S7+nanosleep
S9	S8+futex	S8+unknown	S8+munmap	S8+futex	S8+unknown	S8+futex
S10	S9+munmap	S9+mmap	S9+unknown	S9+unknown	S9+mmap	S9+ppoll
S11	S10+unknown	S10+munmap	S10+mmap	S10+mmap	S10+munmap	S10+recvfrom
S12	S11+mmap	S11+sendto	S11+sendto	S11+munmap	S11+ppoll	S11+munmap
S13	S12+mprotect	S12+recvfrom	S12+recvfrom	S12+ppoll	S12+recvfrom	S12+unknown
S14	S13+sysdigevent	S13+io_getevents	S13+io_getevents	S13+recvfrom	S13+sendto	S13+sendto
S15	S14+io_getevents	S14+ppoll	S14+ppoll	S14+sendto	S14+io_getevents	S14+mmap
S16	S15+sendto	S15+getrusage	S15+getrusage	S15+io_getevents	S15+getrusage	S15+io_getevents
S17	S16+recvfrom	S16+sched_getaffinity	S16+sysdigevent	S16+mprotect	S16+sched_getaffinity	S16+pread



(a) Frequency vs. attack severity.



(b) Sequence of system call vs. bag size.

Figure 4: Performance metrics for two different proposed methods.

Selecting the 82% threshold is essential to guaranteeing precise detection of malicious activities. Adjusting this setting involves finding a balance: although a lower threshold could cause benign behavior to be mistaken for malicious activity, a higher threshold would increase precision at the expense of the system's capacity to recognize subtle attacks. As such, we recognize that this threshold is application-dependent and may have an effect on the overall performance of the system. Changing this threshold affects the system's capacity to distinguish between benign and malicious activity, so it's not just a matter of preference. The 82% level was selected after thorough consideration in order to maximize the system's capacity to correctly detect XSS assaults while lowering the possibility of false positives. This degree of flexibility is essential to guaranteeing our system's efficacy in a variety of usage scenarios and threat environments.

Our approach uses unique sequences from benign profile modifications in Step 3(d) of Fig. 2 to provide a baseline for system calls. More specifically, we use methods like sliding window and Bag of System Calls (BoSC) techniques to produce system call sequence samples during the simulation of a Cross-Site Request Forgery (CSRF) attack. We use a strict criterion to distinguish between benign and harmful actions: behaviors that match the established benign reference by more than 80% are considered benign, while those that fall short of this threshold are classed as possible malicious. During CSRF attack simulations, this method guarantees a stable and flexible system that can recognize and classify system call sequences with accuracy. In order to simulate a more sophisticated threat, specifically a SQL injection (SQLi) at-

tack, it is required that the attacker has root access and database knowledge. We compare system calls made throughout sessions with a retrained root user to closely examine variances in order to identify potential breaches and unauthorized image insertions inside the container. During these sessions, any deviations or anomalies from the standard system call sequences are important indicators of any security lapses, demonstrating the system's capacity to identify and counteract SQLi attacks.

Monitoring the activity of each pod through the Kubernetes dashboard involves utilizing the Seccomp (Kubernets.io, 2022) tool located in the kube-system namespace on the master node. This tool generates log files that meticulously record the various requests made to a pod, including operations such as GET and POST. The primary purpose of these logs is to facilitate the swift identification of suspicious activities, with a particular emphasis on detecting any attempts to access or inject unauthorized content into a pod, especially when operating within the default namespace. This process aligns with Step 1(d) of the overarching framework depicted in the corresponding figure. (Step 1(d) of Fig. 2).

4 IMPLEMENTATION

This section describes the implementation details of our solution.

Implementation Setup. Our Docker containers are running on the host operating system seed-Ubuntu 20.04. Minikube v1.28.0 (minikube, 2023) is utilized to deploy our containers in Kubernetes. With the help of Minikube, we can show Kubernetes on a local computer. We had to generate our own dataset for each of the attack, because there is no dataset that contains the system calls made a container following our attack pathways.

Dataset Generation and Pre-processing. We use Sysdig (Sysdig, 2022) for system call frequency, providing comprehensive container information updated every second after a simulated attack. For system call sequence, we employ strace (Kerrisk, 2022), logging newly created calls during any container activity. Minikube Dashboard aids RBAC attack data collection, offering detailed container information. The dashboard logs Kubernetes API requests, and Seccomp (Kubernets.io, 2022) (Docker, 2022) generates log files automatically on the master node and pods. Python v3.10.1 is used for data pre-processing, capturing frequency data for benign and malicious actions. Focusing on approximately twenty system calls with a significant deviation from normal behavior, we filter out extraneous data from strace and sysdig tools, including system calls' arguments, timestamps, and other details, for both frequency and order of syscalls.

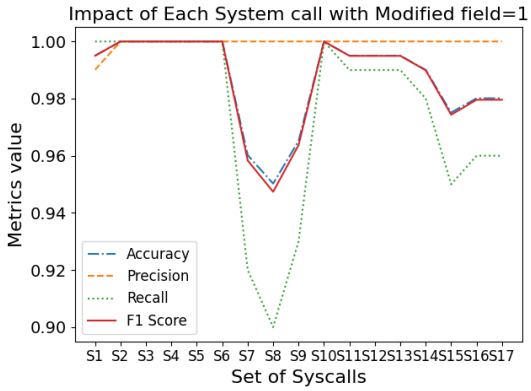
5 EXPERIMENTS & RESULTS

This section presents our experimental results.

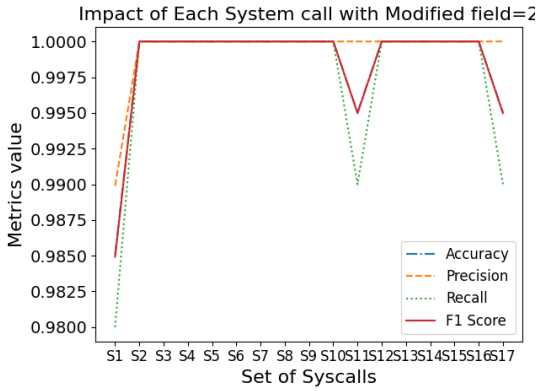
Detection Using Frequency. The detection technique's performance improves with the intensification of the attack, increasing the malicious workload from infecting one to six fields (as shown later). Fig. 4a illustrates that performance metrics (accuracy, precision, recall, and f1 score) increase as the number of infected fields grows, indicating improved detection as attack severity rises. We limit our attack severity to infecting six fields since our detection method consistently performs well under these conditions. Fig. 4a is derived from the top k system calls, determined by their percentage change from benign to malicious activity, showcasing significant shifts with varying attack intensity. Six graphs correspond to different workload lengths, and values along the X-axis are explained in Table 1.

Inference from Fig. 5a suggests optimal performance up to set $S6$ for all metrics. System calls like `io_submit`, `sched_yield`, and `nanosleep` are identified as reducing the effectiveness of frequency-based detection. The same inference applies to the remaining five graphs. The top ten to twelve most significant system calls, demonstrating regular frequency, are selected to build the graph in Fig. 4a.

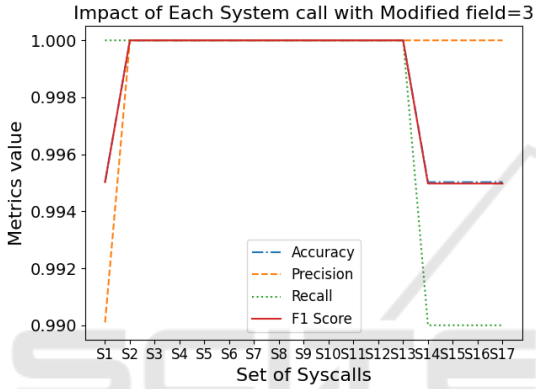
For the second experiment, we mainly use the system calls `write`, `fdatasync`, and `io_submit` in order to distinguish between high workload and an attack. We selected these three because, according to Table 1, they alter the most significantly over the attack severity range of 1 to 6. We run our application 120 times while injecting malicious code via XSS attacks during half of them. We perform various tasks, such as profile view, friend request, direct message, timeline post etc. Through behavioral assessment, we find that during malicious code injection, the event number difference between consecutive events exceeds 40,000, while their time difference remains below 9,99,999 microseconds. It's important to note that these data may vary between different system calls. However, in our scenario, this criterion effectively identified 52 out of 60 malicious operations and 47 out of 60 benign activities for these three system calls. This minimal detection mechanism works well when syscall frequencies become uniform during heavy workloads, aiding in distinguishing XSS attacks alongside frequency-based detection.



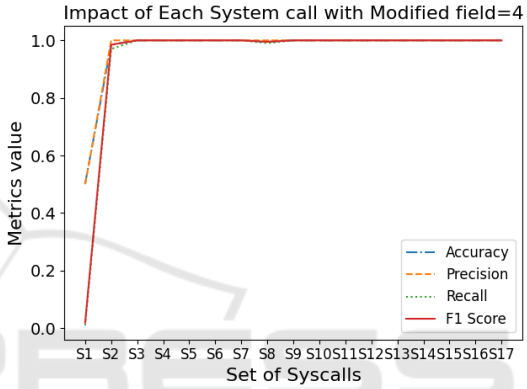
(a) Attack severity=1.



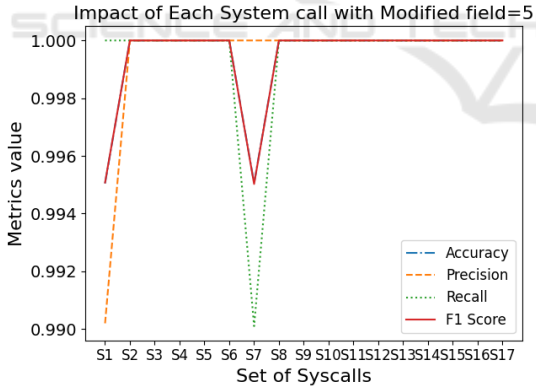
(b) Attack severity=2.



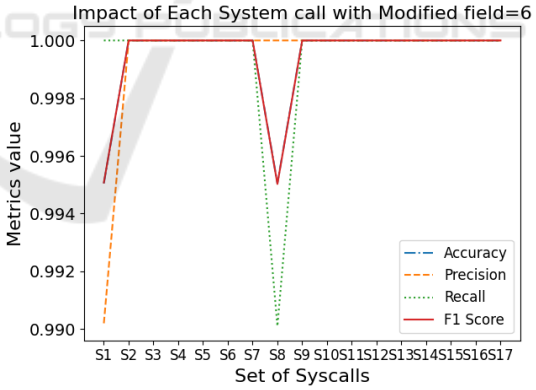
(c) Attack severity=3.



(d) Attack severity=4.



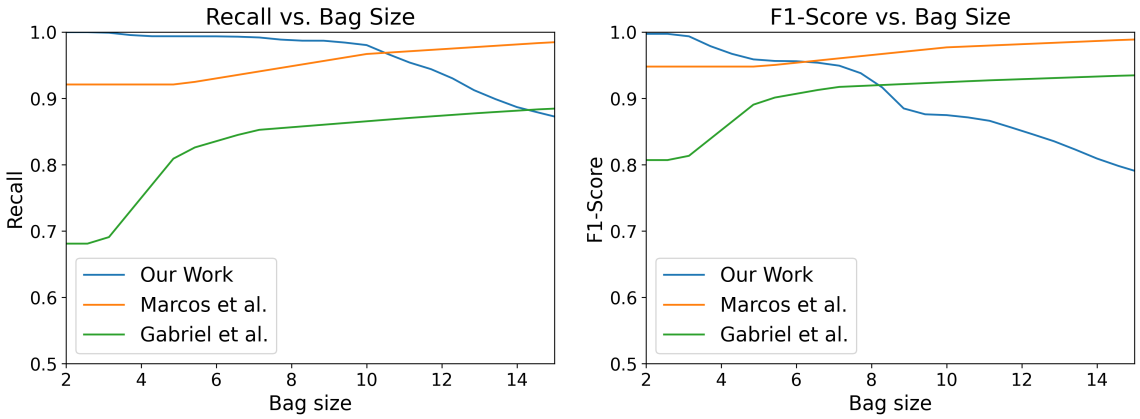
(e) Attack severity=5.



(f) Attack severity=6.

Figure 5: Finding key group of system calls for different intensity of attack.

Detection Using Sequence. When an XSS attack occurs within a container, we have employed the BoSC approach in our experiment. We have experimented with various bag sizes to find the smallest bag yet it provides the greatest performance scores. In our example, we see a sharp decline in the efficacy of our detecting method as we increased bag sizes. In light of our findings, a bag of size between two and six is advised. Many combinations are possible if the bag size is more than six. It is thus exceedingly challenging for our method to discern between legitimate and malicious activities in a container. Also, the higher the bag size, the longer it takes for our suggested technique to determine whether a given activity is benign or malevolent (Hofmeyr et al., 1998). Fig. 4b shows the result of the performance metrics of detection us-



(a) Comparing recall score. (b) Comparing F1 score.

Figure 6: Comparison with (Castanhel et al., 2021; Cavalcanti et al., 2021).

ing sequence.

Detection Using Event Logs. We continually monitor each pod's condition in the master node using Minikube Dashboard. For our RBAC attack, where a Kubernetes API request is made to the master node, we inspect event logs of the kube-apiserver pod in the kube-system namespace (Kubernetes, 2023). Our focus is on the type of API request, the service account token making the request, and the response content. By analyzing these factors, we determine the API request's privilege. If it is privileged but the service account token belongs to the default namespace, we flag it as suspicious.

To validate our results, we compare them with prior research, assessing their performance against our own. Fig. 6a and Fig. 6b illustrate this comparison. In our case, a bag size of 2–10 yields satisfactory performance metrics. Unlike other studies, our performance metrics decline as the bag size increases. This finding is significant; excessively decreasing the bag size limits the variety in our standard sequence, impairing our detection technique, while increasing the bag size above 10 escalates computing costs.

6 DISCUSSION

Threats to Validity. Recognizing and examining any risk that might compromise the validity of our research and affect the dependability and applicability of our findings is crucial. One notable threat to internal validity is the possibility of mixing up malicious activity with high workload in case of using frequency to detect malicious code injection. To mitigate this, we conduct another assessment that is described in the last paragraph of section 3.2. Another threat can be considering unnecessary system calls that may degrade the performance of our detection mechanism. We address this issue by identifying key group of system calls which is shown in Fig. 5

Scaling for Larger Cloud Workloads. Giving careful thought to scaling our solution for bigger cloud environments is important. Since the scope of our study was limited, we had to rely on using a smaller cluster built on Minikube. But an organized strategy is needed to extrapolate our findings to bigger cloud infrastructures. Our solution's scalability depends on effective resource allocation, reliable communication protocols, and load balancing techniques. Subsequent studies should focus on refining these elements to guarantee smooth scaling.

Considering Other Threats. Our approach is effective against Cloud Native Computing Foundation (CNCF) threats, but it is also flexible and resilient to different threat environments. Since our solution includes analyzing log files from Kubernetes dashboard, any suspicious activity in master node by a user in default namespace can be detected. Moreover, it is evident from previous research work ((Abed et al., 2015)) that BoSC technique is not limited to detect only malicious code injection in containerized environment.

Platform Independence. A key component of our solution's broad use is making sure that it can be transferred to other containerized environments as well. While our study focuses on a specific container orchestration platform (e.g., Kubernetes), methods that are used in our study can be mapped to other container orchestration tools like Red Hat OpenShift(Hat, 2023), Docker Swarm(Docker, 2023), etc. The only requirement is that these orchestration tools allow runtime monitoring of log files by collection of var-

ious information. As we collect system call parameters (e.g., frequency, sequence) from operating system level, using frequency and sequence to detect malicious code injection should work irrespective of container orchestration tools.

7 CONCLUSION

In this work, we presented an innovative approach for detecting malicious code injection in containerized systems. Our approach makes use of detailed logs that are sourced from the Kubernetes dashboard, along with an in-depth examination of the names, frequencies, and sequences of system calls. Our method helps create a more thorough and reliable detection system for malicious activity inside of containers by merging these many information sources.

Superior accuracy was demonstrated by our method in differentiating between benign and malicious activities. This achievement can be credited to the careful analysis of system call patterns, which enables us to identify modest but noteworthy deviations suggestive of possible security risks. By means of the incorporation of state-of-the-art techniques, our methodology outperforms current approaches as reported in the literature. Our method's ability to pinpoint the key system calls linked to malicious activity is one of its main advantages. Our method not only improves detection precision by identifying these crucial system calls, but it also offers important information about the particular attack vectors that attackers use.

However, our work has some limitations. While we focused on known attack pathways leading to code injection, there may be other methods not considered. Sequence-based system call detection, especially with extensive sequences, might face challenges in timely identifying malicious behavior. We employed the Kubernetes dashboard for RBAC attacks, but automated log tracking is essential for future improvements. Moreover, the current threshold establishment method based on trial and error could be misleading with a small dataset. Although machine learning techniques were not applied in this work, future research aims to explore their general applicability, investigating how parameters impact learning, detection speed, and accuracy to optimize their values for the intended use.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments. This material is based upon work supported by the Natural Sciences and Engineering Research Council of Canada and Department of National Defence Canada under the Discovery Grants RGPIN-2021-04106 and DGDND-2021-04106.

REFERENCES

Abed, A. S., Clancy, T. C., and Levy, D. S. (2015). Applying bag of system calls for anomalous behavior detection of applications in Linux containers. In *IEEE GC Wkshps 2015*, pages 1–5. IEEE.

Aqua (2023). https://www.aquasec.com/news/kubernetes-clusters-under-attack/. [Online, last accessed: Dec 11, 2023].

Brown, P., Brown, A., Gupta, M., and Abdelsalam, M. (2022). Online malware classification with system-wide system calls in cloud iaas. In *IEEE IRI 2022*, pages 146–151. IEEE.

Bui, T. (2015). Analysis of docker security. *arXiv preprint arXiv:1501.02967*.

Castanhel, G. R., Heinrich, T., Ceschin, F., and Maziero, C. (2021). Taking a peek: An evaluation of anomaly detection using system calls for containers. In *IEEE ISCC 2021*, pages 1–6. IEEE.

Cavalcanti, M., Inacio, P., and Freire, M. (2021). Performance evaluation of container-level anomaly-based intrusion detection systems for multi-tenant applications using machine learning algorithms. In *ARES 2021*, pages 1–9.

Chelladhurai, J., Chelliah, P. R., and Kumar, S. A. (2016). Securing docker containers from denial of service (dos) attacks. In *IEEE SCC 2016*, pages 856–859. IEEE.

CNCF (February 10, 2022). `https://www.cncf.io/reports/cncf-annual-survey-2021/`. [Online, last accessed: May 1, 2022].

Cuen, L. (Mar 12, 2018). Monero mining malware attack linked to Egyptian telecom giant. [Online, last accessed: February 17, 2023].

Docker (2022). Seccomp security profiles for Docker. `https://docs.docker.com/engine/security/seccomp/`. [Online, last accessed: October 11, 2022].

Docker (2023). Docker swarm. https://docs.docker.com/engine/swarm/. Online, last accessed: Feb 19, 2024.

DockerHub (2022). `https://hub.docker.com/`. [Online, last accessed: November 23, 2022].

Gerzi, E. (2022). Compromising Kubernetes Cluster by Exploiting RBAC Permissions. [Online, last accessed: May 6, 2022].

Hat, R. (2023). Red hat openshift container platform. https://www.redhat.com/en/technologies/cloud-computing/openshift/container-platform. Online, last accessed: Feb 19, 2024.

Hofmeyr, S. A., Forrest, S., and Somayaji, A. (1998). Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180.

Jang, S., Song, S., Tak, B., Suneja, S., Le, M. V., Yue, C., and Williams, D. (2022). Secquant: Quantifying container system call exposure. In *European Symposium on Research in Computer Security*, pages 145–166. Springer.

Kerrisk, M. (2022). Strace Documentation. https://man7.org/linux/man-pages/man1/strace.1.html. [Online, last accessed: May 28, 2022].

Kuang, L. and Zulkernine, M. (2008). An intrusion-tolerant mechanism for intrusion detection systems. In *ARES 2008*, pages 319–326. IEEE.

Kubernetes (2023). https://kubernetes.io/docs/concepts/overview/components/. [Online, last accessed: May 14, 2023].

Kubernets.io (2022). Restrict a Container's Syscalls with seccomp. https://kubernetes.io/docs/tutorials/security/seccomp/. [Online, last accessed: August 10, 2022].

Lancini, M. (June 30, 2020). The Current State of Kubernetes Threat Modelling. https://blog.marcolancini.it/2020/blog-kubernetes-threat-modelling/. [Online, accessed May 16, 2023].

Lee, H., Kwon, S., and Lee, J.-H. (2023a). Experimental analysis of security attacks for docker container communications. *Electronics*, 12(4):940.

Lee, K., Lee, J., and Yim, K. (2023b). Classification and analysis of malicious code detection techniques based on the apt attack. *Applied Sciences*, 13(5):2894.

Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., and Zhou, Q. (2018). A measurement study on linux container security: Attacks and countermeasures. In *ACSAC 2018*, pages 418–429.

minikube (2023). https://minikube.sigs.k8s.io/docs/start/. [Online, last accessed: May 15, 2023].

Redhat (2023). State of kubernetes security report 2023. https://www.redhat.com/en/resources/state-kubernetes-security-report-2023.

Sarkale, V. V., Rad, P., and Lee, W. (2017). Secure cloud container: Runtime behavior monitoring using most privileged container (mpc). In *IEEE CSCloud 2017*, pages 351–356. IEEE.

seed-labs 2.0 (2022a). Cross-Site Request Forgery Attack Lab. https://seedsecuritylabs.org/Labs_20.04/Web/Web_CSRF_Elgg/. [Online, last accessed: May 3, 2022].

seed-labs 2.0 (2022b). Cross-Site Scripting Attack Lab (Elgg). https://seedsecuritylabs.org/Labs_20.04/Web/Web_XSS_Elgg/. [Online, last accessed: May 4, 2022].

seed-labs 2.0 (2022c). SQL Injection Attack Lab. https://seedsecuritylabs.org/Labs_20.04/Web/Web_SQL_Injection/. [Online, last accessed: May 3, 2022].

SeedLab (2022). Seed lab documentation. https://seedsecuritylabs.org/labs.html. [Online, last accessed: May 2, 2022].

Son, S., McKinley, K. S., and Shmatikov, V. (2013). Diglossia: detecting code injection attacks with precision and efficiency. In *ACM CCS 2013*, pages 1181–1192.

Song, S., Suneja, S., Le, M. V., and Tak, B. (2023). On the value of sequence-based system call filtering for container security. In *IEEE/ACM UCC 2023*, pages 296–307. IEEE.

Souppaya, M., Morello, J., and Scarfone, K. (2017). Application container security guide. Technical report, National Institute of Standards and Technology.

Sultan, S., Ahmad, I., and Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead. *IEEE access*, 7:52976–52996.

Sysdig (2022). Sysdig Documentation. https://docs.sysdig.com/en/. [Online, last accessed: June 15, 2022].

Tunde-Onadele, O., He, J., Dai, T., and Gu, X. (2019). A study on container vulnerability exploit detection. In *IEEE IC2E 2019)*, pages 121–127. IEEE.

VS, D. P., Sethuraman, S. C., and Khan, M. K. (2023). Container security: Precaution levels, mitigation strategies, and research perspectives. *Computers & Security*, page 103490.

Wong, A. Y., Chekole, E. G., Ochoa, M., and Zhou, J. (2023). On the security of containers: Threat modeling, attack analysis, and mitigation strategies. *Computers & Security*, 128:103140.

Yarygina, T. and Otterstad, C. (2018). A game of microservices: Automated intrusion response. In *DAIS 2018*, pages 169–177. Springer.