Detecting Malicious Code Injection by Monitoring Container Activities

Author AAA and Author BB
Affiliation
Email: AA@cse.buet.ac.bd

Abstract—**The adoption of Docker and Kubernetes has increased along with the rising demand for container technologies. Recent technological assessments indicate that containers are now widely used. This sort of isolated environment's implementation and management pose concerns regarding the security of such systems. Our research attempts to identify risks in a container environment using various system call characteristics (name, frequency, sequence, etc.) and to differentiate between anticipated and unexpected behavior (possible threats).**

Keywords: Instrusion, Container, Kubernetes, Microservice, Malicious Code Injection.

I. Introduction

The use of containers has substantially increased over the past several years as well as their popularity. According to Grand View Research [1], The worldwide container market had a value of $2.05 billion in 2020, and from 2021 to 2028, it is anticipated to expand at a CAGR of 29.4%. One of the most widely used container platforms, Docker [2], experienced over 11 billion pulls (downloads) from its open repository in 2020. According to a poll performed by the Cloud Native Computing Foundation (CNCF) [3], 92% of respondents were utilizing containers in production in 2020, up from 84% in 2019. In 2021, 57% of enterprises have adapted containers in production, up from 47% in 2020, according to a Flexera poll [4]. Kubernetes has also grown in significance in recent years as a platform for container orchestration because of its scalability, portability, resilience, automation, and big and engaged community. According to a Datadog [5] poll organized in 2019, the use of Kubernetes and other container orchestration solutions would rise by 47% by 2020.

Although the use of container technology is growing every day, it also exposes us to new security threats [6]. For example, in 2020, due to a flaw in the Azure Container Registry (ACR), personal files could be accessed by unauthorized users [7]. Additionally, In 2018, a security researcher revealed that infected systems were mining the cryptocurrency Monero using Docker containers [8]. By injecting malicious code into the containers, the attackers were able to mine Monero using the resources of the servers that had been hacked. In 2019, an attacker compromised Docker Hub, a repository for Docker container images, and introduced malicious malware into a number of images [9]. The hacker was able to acquire a single Docker Hub user's login information, which gave them the ability to insert harmful code into a lot of images.

There are many possible attack scenarios in a container [10]. For example, DoS attack, privilege escalation and container runtime escape attack, remote code execution, malicious code injection etc. One of the most dangerous risks to containerized environments is malicious code injection, which gives attackers the opportunity to access private information, stop operations, or use the container as a launchpad for other assaults [11]. By altering the Dockerfile or other build configuration files, an attacker can introduce malicious code into a container image during the construction process. Additionally, by taking advantage of holes in the application code, an attacker can introduce malicious code into an application that is executing inside a container. For instance, if an input field is not adequately checked, an attacker may introduce harmful code into it. Moreover, to insert malicious code into a container, an attacker can

take advantage of flaws in the host operating system or the container runtime. To recognize and stop security risks that might jeopardize the integrity and confidentiality of applications running in containerized environments, it is important to work with malicious code injection in containers.
Each container in a containerized environment interacts with the kernel through a number of system calls. Depending on the activity taking place inside the containers, these system calls alter. To mitigate such attacks, system call analysis might be a promising solution (as shown in [12]–[14]). However, rather than concentrating on specialized solutions for certain attack pathways, the current solutions offer an generalized technique to prevent attacks on containers. For instance, several attacks like XSS, CSRF, SQLi, RBAC attack, etc. can be used to inject malicious code and each of these attacks takes a different path. These various attack methods make use of various container and kubernetes vulnerabilities. Hence, for these many attack pathways, applying a single generalized solution may not produce the desired results. Hence, a combination of diverse detection mechanisms that have not been used in earlier research should be used to counter these threats.

Each container operates independently in an isolated environment with its own set of processes, file systems, and network connections in a containerized environment. Here isloated environment refers to the fact that each container is isolated from the host operating system by a feature known as sandboxing, which prevents it from accessing resources outside of its own environment unless specifically given permission. Containers can also be started and terminated fast and simply since they are meant to be ephemeral. As a result, tracking and monitoring container activity over time may be challenging, further complicating attempts to identify intrusions. Along with this, containerized environments provide a quick scaling up and down capability to adapt to demand changes. In order to keep up with the changing environment, a centralized detection technique may need to be scalable fast. Due to these factors, it may be challenging to identify security risks and take appropriate action because typical intrusion detection systems might not be able to see what is happening in each individual container.
We did not find any existing research work that focuses on preventing malicious code injection in containers for multiple potential pathways. This might entail utilizing methods like runtime monitoring, behavior-based analysis, etc to quickly identify and address these kinds of security problems.

In this paper, we suggest combining a number of strategies to address the multi-path nature of malicious code injection and, as a result, offer defense against various forms of malicious code injection attacks. Moreover, we do not simply provide comparison among some machine learning algorithms as prior works such as [12], [14], rather present a in-depth look for selection of some key system calls to justify the result of our proposed method.
We made advantage of many attack scenarios that the CNCF Finance User Group supplied [15]. Our main contributions in this research are:

- We monitored container activities at both k8s and syscall level to detect malicious code injection attack. As a result, layered monitoring makes it feasible to identify an attack in a different layer if an attacker manages to get past a single layer of a container without being detected.
- We determined the impacts of major attack steps of malicious code injection attacks on the behavior of container activities (e.g., frequency of syscalls, sequence of syscalls) – while considering major attack steps from the CNCF framework. We cover all possible routes that this study provides to defend against a wide-range of malicious code injection attacks.
- We separated key group of system calls from all the system calls we observed that show significant change during an attack. Frequency was employed to filter out useless system calls while detecting malicious activity. Because we were no longer concerned with pointless system calls, our mechanism worked better and was more effective.
- We evaluated our proposed method for 4 different attack paths. Different number of containers

were used for simulating attack on different paths. While working with frequency, we filtered out 18 system calls out of 28. But in case of sequence of system calls we considered all these system calls. From the results and graphs we generated using our proposed method, we can observe decent scores in performance metrics (more than 90%).

The rest of the paper is organized as follows. In Section II, we discussed some existing work on container security. Section III presents the details of our proposed solution. Section IV contains our implementation setup and dataset handling. In Section V, we discussed the outcome of our experiment. Finally, we conclude the paper in Section VI.

II. Background and Literature Review

This section provides an overview of the existing works on addressing the possible solutions to different attacks in containerized environment and the terminologies required to comprehend them. To comprehend our suggested approach and implementation, it will be required to understand the terms provided in this part.

A. Terminologies

A software architectural paradigm called microservices divides programs into a number of tiny, unrelated, and loosely linked services. The ability to deploy several microservices on a single physical server is made possible by virtuallization, which may be utilized to build a virtual environment. This might save hardware costs, improve resource use, and increase scalability.

Virtuallization techniques can be divided into two types [16]:

- Container-based virtualization
- Hypervisor-based virtualization

Between them, due to its many advantages over virtual machines (VMs), such as their light weight, speed, ease of deployment, ability to better use resources, and support for version control, containers are a more realistic alternative for microservices [10].

On the other side, Kubernetes is an open-source platform for container orchestration that streamlines the management, scaling, and deployment of containerized applications. For managing containerized applications, Kubernetes offers a comprehensive set of functions, including load balancing, self-healing, automatic deployment and scaling, and rolling upgrades. A master node and one or more worker nodes are part of the cluster of nodes that make up the Kubernetes architecture. To manage the containers that are operating on each worker node, each worker node runs a container runtime, such as Docker. The Kubernetes API server, etcd, kubelet, kube-proxy, and different add-ons and plugins are just a few of the controllers and components that Kubernetes employs to handle the deployment, scaling, and updating of applications. Fig. 2 summarizes the structure and working principle of kubernetes.

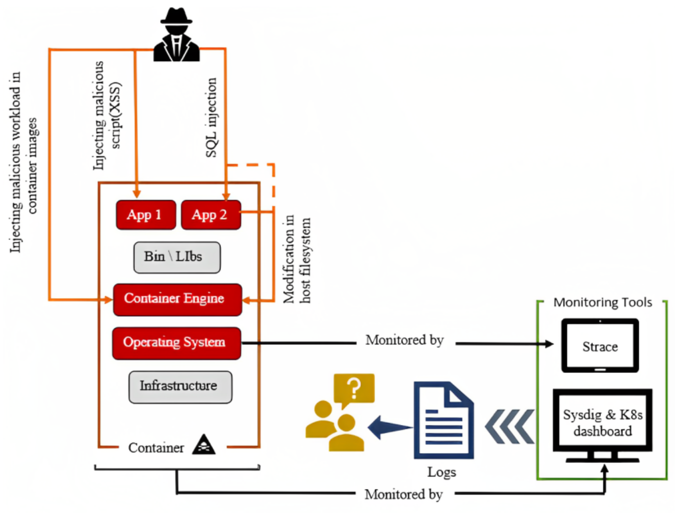


Fig. 1: Proposed method to monitor different layers of container through different monitoring tools

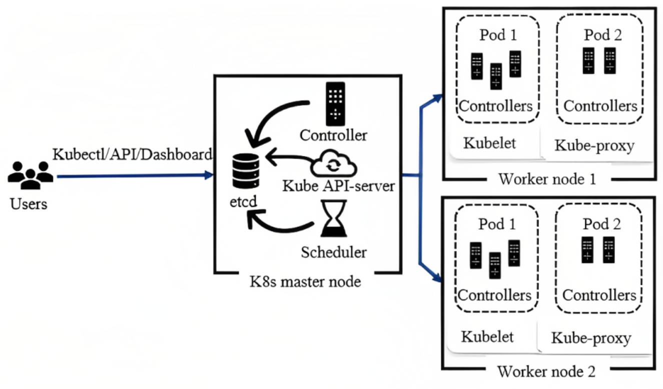


Fig. 2: Structure and working principle of Kubernetes

B. Existing works

On detection of attack in container, several works are already in existence. Neither all of them have a specific connection to containerized environments nor all of them are related to malicious code injection detection in container.
The existing study on intrusion detection system at the container level is summarized in this paragraph. For Linux containerized systems, Abed *et al.* [12] suggested a host-based intrusion detection solution. Cavalcanti, Inácio and Freire [14] proposed a container-level anomaly-Based intrusion detection systems for multi-tenant applications using machine learning algorithms. In this literature, the researchers mainly worked on four different vulnerabilities - Authentication Bypass, DoS Overflow, System User Privilege Escalation and Integer Overflow. Lin *et al.* [17] gathered a dataset of attacks, including 223 vulnerabilities that work on the container platform, and used a two-dimensional attack taxonomy to categorize them. Next they used 88 common attacks fished out of the dataset to assess the security of the current Linux container technology. Chelladhurai *et al.* [18] addressed crucial Docker container security challenges as well as related work that is being done in this field. Also, they have suggested security approaches and algorithms to deal with problems caused by DoS attacks in the Docker container technology. Tunde-Onadele *et al.* [19] proposed detection mechanism using frequency of selective system calls that can detect more than 20 out of 28 tested vulnerability exploits. They only worked with 4 specific system calls and the reasoning for choosing only these system calls was absent. The articles described above all dealt with various types of attacks, including SQL injection utilizing a MySQL container image, system user privilege escalation, authentication circumvention, remote code execution (XSS, XEE), DoS attack, etc. Although some of these attacks are variations on malicious code injection in some way, they are not the only means of doing so. In addition to that, for all these attack, all of them proposed a common solution - mostly using sequence of system calls. Unlike our work, they did not propose attack specific detection mechanism.
Yarygina and Otterstad [20] proposed a a flexible, cost-conscious intrusion response system for microservices that employs a game-theoretic method to automatically respond in real time to network threats. Thanh Bui [16], in his paper, gave us an analytical view of container security. His research focuses on two areas: (1) Docker's internal security like different isolation techniques and (2) Docker's interactions with the Linux kernel's security features. Souppaya *et al.* [11] discusses the possible security issues raised by the usage of containers and offers suggestions for resolving them. Sarkale, Rad and Lee [21] suggested a new security layer using the Most Privileged Container (MPC). The newly implemented MPC layer implements privilege-based access control and grants permissions for resource access based on rules and the security profiles of user processes for containerized application. All these papers provided deep idea about the built-in or customized security measures of containers and how one can enable them to secure container from different kinds of attacks. But unlike us, none of them tested their suggested methodology, let alone worked on a specific attack signature.
Castanhel *et al.* [13] and Hofmeyr and Anil Somayaji [22] worked on detecting anomaly using sequence of system calls and different machine learning algorithms. In their papers, they provided how the change of length of system calls can impact intrusion detection using sequence of system calls. They did not work on any particular attacks that can finally lead to malicious code injection.
There are also several works that have nothing to do with containerized environments particularly. Yet, the solutions they provide may also be employed in containers. For network intrusion detection systems (NIDS) that use numerous independent components, Kuang and Zulkernine [23] suggested an intrusion-tolerant approach. The technique allows the IDS to withstand component failure brought on by intrusions and keeps track of the detection units and the hosts on which they are hosted. A copy of the failed IDS component is installed to replace it as soon as it is found, and the detection service is then resumed. Son *et al.* [24] designed, implemented and evaluated, DIGLOSSIA, a novel runtime tool for the accurate and efficient detection of code injection threats.

III. Methodology

From the perspective of the attacker, there are several ways to attack a container-based virtualization system, such as using virtualization to steal users' private data, initiating attacks by injecting malicious workload, or escalating the intrusion to numerous VM instances [13].

The CNCF Financial User Group released [15], in January 2020, documentation and outcomes of an in-depth threat modeling exercise performed against a generic Kubernetes cluster. Their work provided a detailed view of several attack paths by which an attacker could exploit configuration vulnerabilities within Kubernetes to achieve specific goals.

A. Approach Overview

Our work contributes to the mitigation of all the attack paths that finally leads to malicious code execution in container. Fig. 3 contains the classification of these attack paths according to distinct detection mechanisms.

There are two basic categories of malicious workload detection mechanisms. One is based on system calls, while the other is based on Kubernetes log files. There are a number of factors that need to be taken into account when employing a system call-based method, including the name, sequence, and frequency of the system call. For the suggested approach to function, these parameters might need to be combined.

Firstly, many attacks (XSS, CSRF, SQLi, RBAC attack, etc.) that adhere to the attack pathways offered by CNCF were simulated, and for each path, detection techniques were carried out. By examining the names, frequency, and sequences of system calls, the distinction between benign and malicious activities may be made.

Our operating system, Seed-Ubuntu, is utilized for the aim of simulating an attack. Reverse engineering tools like Sysdig [25] and Strace [26] are used to gather various system call parameters. The log files for RBAC attacks are collected by the Kubernetes dashboard. Minikube is then used to implement Kubernetes on a local PC.

We acquired enough information to establish a foundation of usual behaviors, which included system call sequences and frequencies that reflected the application's normal functioning as well as the activities when it was under danger or participating in malicious conduct.

To launch an attack using several of the attack pathways, root privileges were needed. In this instance, several new system calls were produced that weren't present during routine operations. These are easily traceable using the names of the system calls, and the Strace tool was employed for this. The Sysdig tool was also used to retrieve the frequency of each system call. Only system calls that significantly changed as a result of malicious behavior were taken into account. In order to work with the sequence of system calls, the BoSc (Bag of System Call) approach was utilized.

B. Modeling Benign Behavior

We purposefully visited several benign profiles to determine the typical frequency of benign activity, and then we used the data to determine the usual frequency (threshold) of each system call. Our dataset required to be divided into a training set and a testing set even though our program did not employ any machine learning techniques. Both benign and malicious data are included in our training set. We identified the top k impactful system calls from the training set based on the frequency at which they changed from benign to malicious activities. The lower bound of the frequency for each system call was determined using the average frequency of each system call from the benign component, and the higher bound was set using the average frequency of each system call from the malicious part. The threshold was established by averaging the upper and lower bounds. We defined a testing sample as benign if more than 82% of the randomly chosen system calls are below this level. We used a trial-and-error methodology to get at this "82 percent" so that we might have the best recall score.

We performed some innocuous profile editing and gathered the standard sequence of system calls for them in order to obtain the standard sequence of system calls. In this context, we applied the Bag of System Calls (BoSC) and sliding window techniques. [12] This method extracts a sliding window from the entire list of system calls that can range in length from 2 to 10, let's say 10. The frequency of each system call in that order was then listed. The

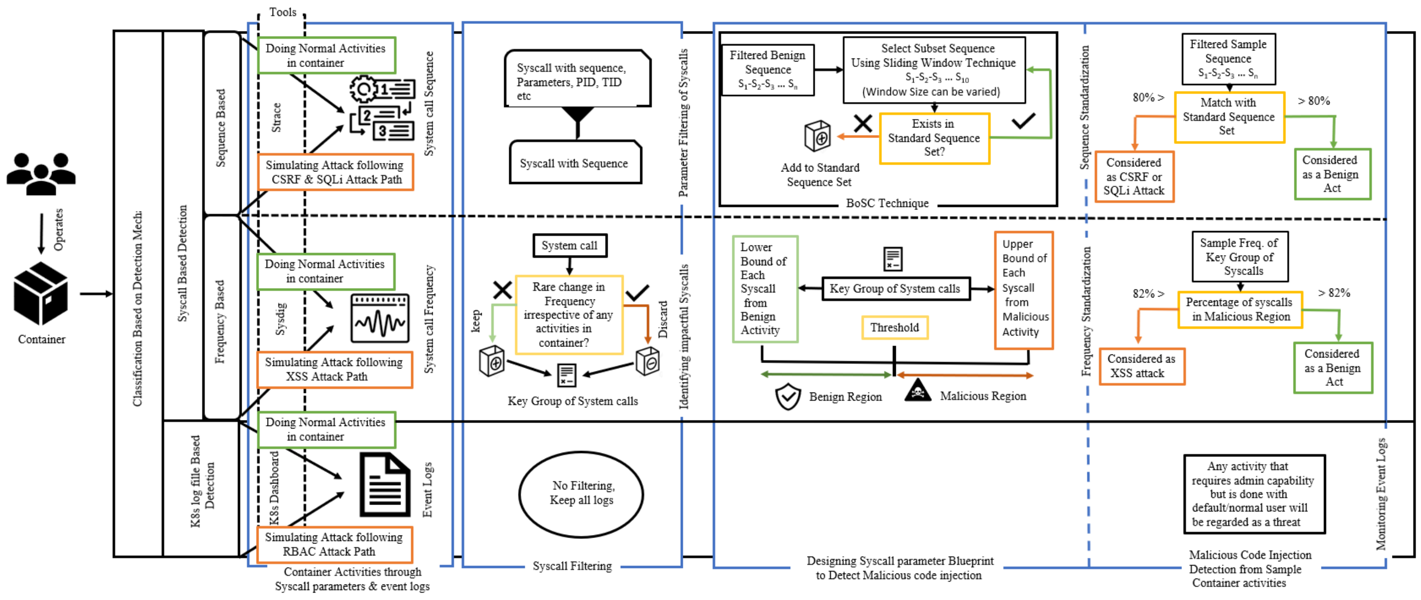

Tools
Container
Operates
Classification Based on Detection Mech.
Syscall Based Detection
Sequence Based
Frequency Based
K8s log file Based Detection
Strace
Sysdig
K8s Dashboard
Doing Normal Activities in container
Simulating Attack following CSRF & SQLi Attack Path
Simulating Attack following XSS Attack Path
Simulating Attack following RBAC Attack Path
System call Sequence
System call Frequency
Event Logs
Container Activities through Syscall parameters & event logs
Syscall with sequence, Parameters, PID, TID etc
Syscall with Sequence
System call
Rare change in Frequency irrespective of any activities in container?
keep
Discard
Key Group of System calls
No Filtering, Keep all logs
Syscall Filtering
Parameter Filtering of Syscalls
Identifying impactful Syscalls
Filtered Benign Sequence S1-S2-S3 ... Sn
Select Subset Sequence Using Sliding Window Technique S1-S2-S3 ... S10 (Window Size can be varied)
Exists in Standard Sequence Set?
Add to Standard Sequence Set
BoSC Technique
Lower Bound of Each Syscall from Benign Activity
Key Group of System calls
Upper Bound of Each Syscall from Malicious Activity
Threshold
Benign Region
Malicious Region
Designing Syscall parameter Blueprint to Detect Malicious code injection
Sequence Standardization
Frequency Standardization
Filtered Sample Sequence S1-S2-S3 ... Sn
Match with Standard Sequence Set
80% >
> 80%
Considered as CSRF or SQLi Attack
Considered as a Benign Act
Sample Freq. of Key Group of Syscalls
Percentage of syscalls in Malicious Region
82% >
> 82%
Considered as XSS attack
Considered as a Benign Act
Any activity that requires admin capability but is done with default/normal user will be regarded as a threat
Monitoring Event Logs
Malicious Code Injection Detection from Sample Container activities

Fig. 3: Classification of attack path based on detection mechanism and their corresponding flow diagram.

frequency list can look something like this if there are 20 separate system calls: [0, 1, 1, 1, 0, 0, 0, 0, 1, 2, 2, 0, 0, 0, 0, 0, 1, 0, 0, 1]. Following that, we take a new sequence by sliding one window of system call, and we create a new frequency list using the new system call sequence. We now add that list to our list of unique bag of sequences unless it has already been produced, in which case we disregard it.
Last but not least, we recorded 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.

C. Simulating the attacks

Attack Overview: From the attack path provided by CNCF, four different attack paths can be identified that finally lead to malicious code injection.

To simulate each of the attack path, we used different kinds of attack on container. For Fig. 4, SQLi attack was implemented. XSS attack follows the path of Fig. 5. For Fig. 6 and Fig. 7, CSRF and RBAC attack were simulated. Apart from the RBAC attack, the others were implemented using the attacks demonstrated in SEED-LABS 2.0 [27].

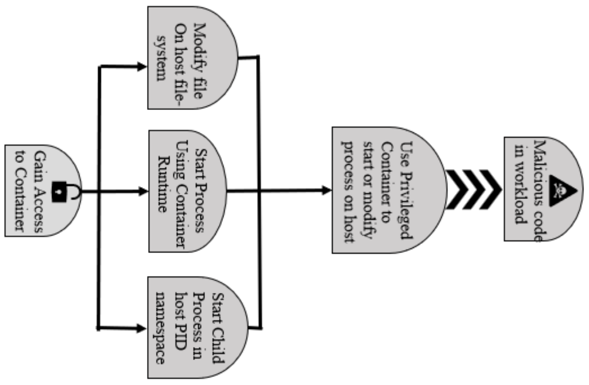

Gain Access to Container
Modify file On host file-system
Start Process Using Container Runtime
Start Child Process in host PID namespace
Use Privileged Container to start or modify process on host
Malicious code in workload

Fig. 4: SQL injection attack

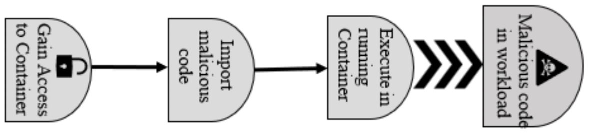


Fig. 5: XSS attack

For four alternative attack pathways, we simulated four different types of attacks. We utilized a built-in website that functions like a social networking platform in seed-lab 2.0 for two of them (XSS and CSRF). Certain users on this website have the ability to edit their profiles, view the profiles of

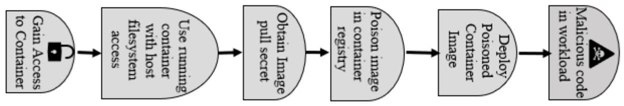

Gain Access to Container
Use running container with host filesystem access
Obtain Image pull secret
Poison image in container registry
Deploy Poisoned Container Image
Malicious code in workload

Fig. 6: CSRF attack

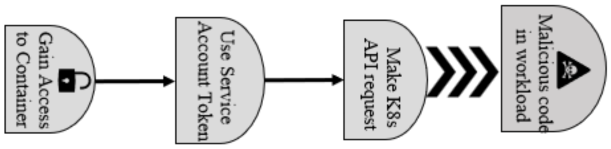


Fig. 7: RBAC attack

others, make friend requests, etc. And for SQLi, a company website with a database where employees can see their profiles and make changes to certain fields. There is also an admin who has access to all the information related to his staff.

1) CSRF attack: The bengin social networking website was combined with a malicious container that included malicious images in this attack. A user's profile is modified without his consent when he accesses the malicious website installed in this harmful container. This is due to a concealed POST request to the benign container in the image of the malicious container. [28]

2) XSS attack: In this approach, one of the website's users(attacker) infects the benign website. On his profile area, he developed a script (malicious workload) that may influence other people's profiles. This attacker's profile is changed without his knowledge when a decent user sees it. [29]

3) SQLi attack: In order to alter the database tables using SQL injection, the attacker must utilize the login page to insert a malicious query. For the attack to be effective, the attacker must already be familiar with the database table. [30]

4) RBAC attack: A service account with the privilege (or role) of generating containers in the kube-system namespace was used to simulate a RABC or role-based access control attack against Kubernetes. Now that a container has been created in the master node, the attacker may inject a malicious workload into it. In our situation, the malicious container has access to all the service accounts in the kube-system namespace and may steal their secrets. [31]
All of these attacks have their own defenses against them, but for the purposes of our experiment, we purposefully turned them off.

D. Segregation of Malicious code injection from Benign Activities

Prior to now, we established a benchmark for the number of times each system is called during a certain activity (in our case, visiting any profile). Now, we deliberately introduce a hostile workload into a certain profile (XSS attack). By changing the number of modified fields through that malicious script from 1 to 6, we were able to modify the attack's intensity. In each instance, we compared numerous frequency samples to our reference frequency for appropriate behavior. For every sample, if more than 82 of the chosen system calls exceed our usual frequency criterion, it is considered an attack.
We already have a list of distinct sequences from the benign profile edit while dealing with the sequence of system calls. We created a large number of samples of system call sequences while simulating a CSRF attack. The same bag of system calls (BoSC) and sliding window techniques were used. Following that, these sequences were contrasted with the normative sequences. We classified an activity as benign if more than 80% of the newly created sequence matches the reference sequence; otherwise, the activity was labeled as malevolent behavior.
As was already established, in order to mimic a SQLi attack, an attacker has to have some knowledge of the tables in a database hosted on the host server. He must have root privileges to do it. Thus, we opened a trace file on the database container and purposefully gained root access to it. Many unique system calls were created in this instance, and we compared them to the typical system calls made by a non-root user. In this instance, there is a presumption, though. We're supposing that we're retaining the sessions of the container's actual root users. So, if we discover system calls done by a root user but not when that user was really logged in, we may conclude that our system has been maliciously breached, allowing an attacker to insert a malicious image into a container.

The activity in each pod(container) can be obtained through the Kubernetes dashboard. This is because Seccomp [32] tool is already installed in every pod in the kube-system namespace that is in the master node. It generates a log file for every pod, and from those log files, we can simply track the requests (GET, POST) made to a certain pod by a user using their service account or by another pod. With the service account token, we can keep track of each service account, and if one attempts to access or inject something (such as a malicious script) into any pod of the master node while using the default namespace, we may flag that service account as suspicious.

IV. Implementation

A. Implementation Setup

Our Docker containers are running on the host operating system seed-Ubuntu 20.04. Minikube v1.28.0 was 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 datasets for each of the attacks because there was no dataset that simultaneously contained the system calls from a container and followed our attack pathways.

B. Dataset Generation

Using the sysdig utility, the frequency of each system call was collected. One may acquire a variety of information about containers using sysdig, such as the processes that are executing within a container, the system calls that are generated within a process, their frequency, and other factors. Every second, Sysdig updates the number of system calls. So, for our investigation, we only recorded the frequencies immediately following a simulated attack.
We utilized the Strace tool to capture the sequence of system calls. Strace, as contrast to Sysdig, only adds freshly created system calls upon any activity done on a container. We launched strace on a target process and downloaded each and every system call made throughout the course of the assault. The outcome was kept in a log file.
Minikube Dashboard was used to gather the data for the RBAC attack. The dashboard includes all of the information about each container, including the processes that are now operating inside of it, its health, the active pods in each namespace, etc. A Kubernetes API request was made to mimic an RBAC attack in the kube-system namespace. As a result, we used the log files from kube-apiserver, one of the pods in the kube-system namespace, to gather data. Seccomp tool [32] [33] is automatically installed to the master node along with all of the pods. The log files for each of these pods are produced by this tool.

C. Pre-processing data

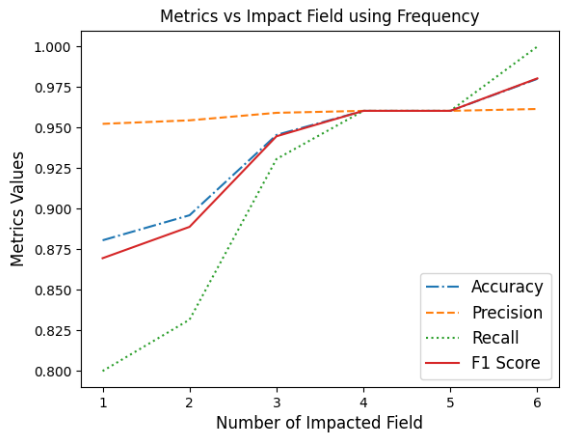
We used Python 3.10.1 to pre-process the data. We gathered frequency data for both benign and malicious actions in order to process the frequency data. Despite the fact that throughout an action, several unique system calls were produced, not all of them were equally affected by an attack. Hence, we only considered system calls that displayed a notable deviation from customary behavior. We discovered that around 20 out of all system calls exhibited this behavioral modification.
We also filtered out any extraneous data gleaned from the strace and sysdig tools, which was applied to both frequency and order of system calls. For instance, system calls' arguments, time-stamps, and other details were removed.

V. Experiment & Result

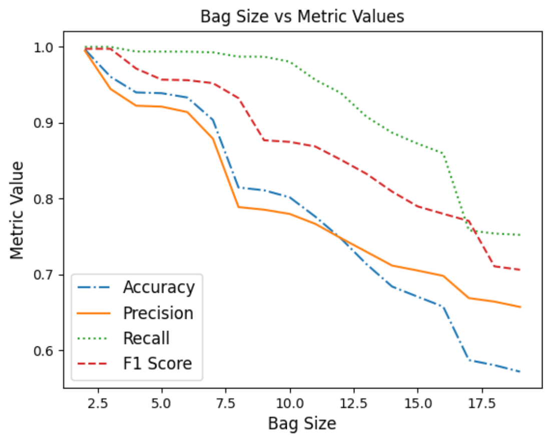
A. Detection using Frequency

We changed the malicious workload size in our experiment to reflect the intensity of each attack in our situation. Our detecting technique is more precise the more severe the attack. Fig. 8a shows the performance metrics of our detection mechanism using frequency.

This graph was not produced entirely using system calls. We initially determined the percentage change of each system call from benign to malicious activity and ordered them according to their percentage change in order to determine the top k system calls. The impact of each system call was then assessed by gradually adding them to the initial set. Six graphs were produced for various workload lengths.

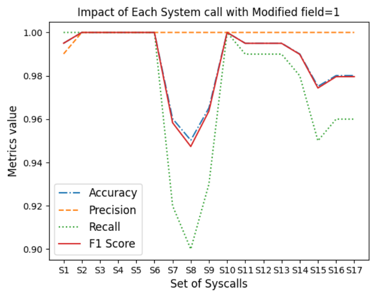


(a) Performance Metrices for detection using frequency with respect to attack severity

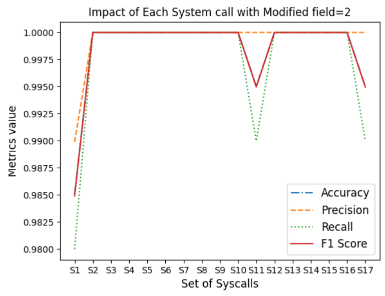


(b) Performance metrics with respect to bag size using sequence of system call for detection

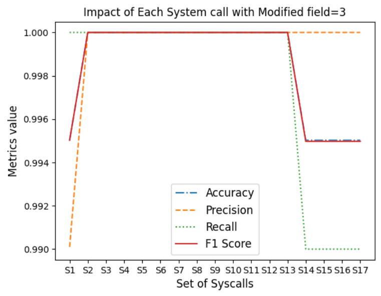
Fig. 8: Performance Metrices for two different proposed method



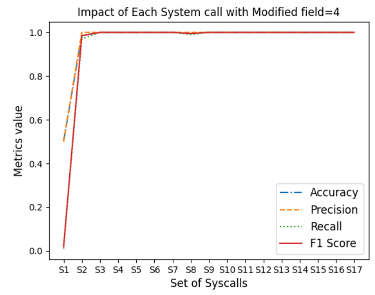
(a) Choosing the best set of system calls for attack severity=1



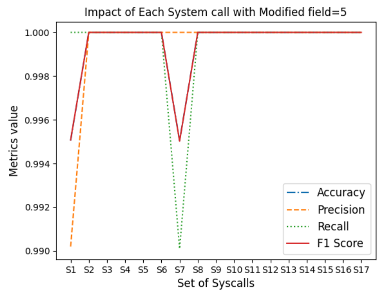
(b) Choosing the best set of system calls for attack severity=2



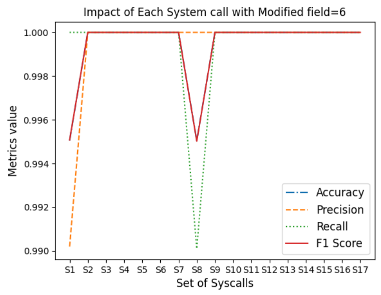
(c) Choosing the best set of system calls for attack severity=3



(d) Choosing the best set of system calls for attack severity=4



(e) Choosing the best set of system calls for attack severity=5



(f) Choosing the best set of system calls for attack severity=6

Fig. 9: Finding key group of system calls for different intensity of attack

Values along X-axis in fig 9a and 9b		
Set	Modified Field=1	Modified Field=2
S1	write	pwrite
S2	S1+fdatasync	S1+write
S3	S2+pwrite	S2+fdatasync
S4	S3+pread	S3+fsync
S5	S4+fsync	S4+io_submit
S6	S5+io_submit	S5+sched_yield
S7	S6+sched_yield	S6+futex
S8	S7+nanosleep	S7+nanosleep
S9	S8+futex	S8+unknown
S10	S9+munmap	S9+mmap
S11	S10+unknown	S10+munmap
S12	S11+mmap	S11+sendto
S13	S12+mprotect	S12+recvfrom
S14	S13+sysdigevent	S13+io_getevents
S15	S14+io_getevents	S14+ppoll
S16	S15+sendto	S15+getrusage
S17	S16+recvfrom	S16+sched_getaffinity

Values along X-axis in fig 9e and 9f		
Set	Modified Field=5	Modified Field=6
S1	write	fdatasync
S2	S1+fdatasync	S1+write
S3	S2+pwrite	S2+pwrite
S4	S3+fsync	S3+do_submit
S5	S4+io_submit	S4+fsync
S6	S5+sched_yield	S5+io_submit
S7	S6+nanosleep	S6+sched_yield
S8	S7+futex	S7+nanosleep
S9	S8+unknown	S8+futex
S10	S9+mmap	S9+ppoll
S11	S10+munmap	S10+recvfrom
S12	S11+ppoll	S11+munmap
S13	S12+recvfrom	S12+unknown
S14	S13+sendto	S13+sendto
S15	S14+io_getevents	S14+mmap
S16	S15+getrusage	S15+io_getevents
S17	S16+sched_getaffinity	S16+pread

Values along X-axis in fig 9c and 9d		
Set	Modified Field=3	Modified Field=4
S1	fdatasync	pread
S2	S1+write	S1+fdatasync
S3	S2+pwrite	S2+write
S4	S3+fsync	S3+pwrite
S5	S4+io_submit	S4+fsync
S6	S5+sched_yield	S5+io_submit
S7	S6+futex	S6+sched_yield
S8	S7+nanosleep	S7+nanosleep
S9	S8+munmap	S8+futex
S10	S9+unknown	S9+unknown
S11	S10+mmap	S10+mmap
S12	S11+sendto	S11+munmap
S13	S12+recvfrom	S12+ppoll
S14	S13+io_getevents	S13+recvfrom
S15	S14+ppoll	S14+sendto
S16	S15+getrusage	S15+io_getevents
S17	S16+sysdigevent	S16+mprotect

We may infer from Fig. 9a that up to the set S6, we were receiving the greatest value possible for all performance metrics. System calls like io_submit, sched_yield, and nanosleep reduce the effectiveness of frequency-based detection. The same inference may be made from the remaining five graphs. Also, we may deduce the top 10–12 most significant system calls from the table. We mostly chose these top 10–12 system calls, which demonstrated regular frequency, to build the graph in Fig. 8a.

B. Detection using Sequence

When an XSS assault occurred within a container, we employed the BoSC approach in our experiment. We experimented various bag sizes to find the smallest bag that yet provided the greatest performance scores. In our example, we saw 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 2 and 6 is advised.
Many combinations are possible if the bag size is more than 6. It is thus exceedingly challenging for our method to discern between legitimate and malicious activities in a container. Also, the longer it takes for our suggested technique to determine whether a given activity is benign or malevolent, the higher the bag size [22]. Fig. 8 shows the

result of the performance metrics of detection using sequence.

VI. Conclusion and Future Work

For identifying anomalies in Linux containers, we provided various detection approaches in this study for various attack vectors. Our suggested method comprises log files from the Kubernetes dashboard and system call frequency, names, and sequences. We were able to discern between malicious and benign behavior with excellent accuracy using all of the aforementioned strategies. Also, we divided the top system calls that exhibit notable variations in response to container activity.

One flaw in our mechanism is that we focused solely on the attack vectors that ultimately resulted in the insertion of malicious code. In addition to them, there could be more options. Sequence detection has another disadvantage over frequency detection in that it might take longer to identify malicious activity if the sequence is too lengthy. We made use of the Kubernetes dashboard to spot RBAC attacks. In our trial, we manually kept track of the dashboard's created log file, which really needed to be automated.

Our next step is to adapt our approach to be more suited for implementation in a real-time intrusion detection system. Also, the mechanism presently chooses some of the settings at random. Future research will focus on analyzing how these parameters affect learning, detection speed, and accuracy in an effort to formalize a method for maximizing their values for the intended application.

References

[1] "," https://www.grandviewresearch.com/, [Online].

[2] "," https://hub.docker.com/, [Online].

[3] "," https://www.cncf.io/, [Online].

[4] "," https://www.flexera.com/, [Online].

[5] "," https://www.datadoghq.com/, [Online].

[6] J. N. Müller, "Static source code analysis of container manifests."

[7] K. Rajapakse, "Azure container registry image scanning," https://faun.pub/azure-container-registry-image-scanning-aaeae84d1c0c, May 18, 2020, [Online].

[8] L. Cuen, "Monero mining malware attack linked to egyptian telecom giant," https://www.coindesk.com/markets/2018/03/12/monero-mining-malware-attack-linked-to-egyptian-telecom-giant/, Mar 12, 2018, [Online].

[9] C. Cimpanu, "Monero mining malware attack linked to egyptian telecom giant," https://www.zdnet.com/article/docker-hub-hack-exposed-data-of-190000-users/, Apr 17, 2019, [Online].

[10] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE access*, vol. 7, pp. 52 976–52 996, 2019.

[11] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," National Institute of Standards and Technology, Tech. Rep., 2017.

[12] A. S. Abed, T. C. Clancy, and D. S. Levy, "Applying bag of system calls for anomalous behavior detection of applications in linux containers," in *2015 IEEE globecom workshops (GC Wkshps)*. IEEE, 2015, pp. 1–5.

[13] G. R. Castanhel, T. Heinrich, F. Ceschin, and C. Maziero, "Taking a peek: An evaluation of anomaly detection using system calls for containers," in *2021 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2021, pp. 1–6.

[14] M. Cavalcanti, P. Inacio, and M. Freire, "Performance evaluation of container-level anomaly-based intrusion detection systems for multi-tenant applications using machine learning algorithms," in *Proceedings of the 16th International Conference on Availability, Reliability and Security*, 2021, pp. 1–9.

[15] M. Lancini, "The Current State of Kubernetes Threat Modelling," https://blog.marcolancini.it/2020/blog-kubernetes-threat-modelling/, June 30, 2020, [Online].

[16] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015.

[17] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418–429.

[18] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, "Securing docker containers from denial of service (dos) attacks," in *2016 IEEE International Conference on Services Computing (SCC)*. IEEE, 2016, pp. 856–859.

[19] O. Tunde-Onadele, J. He, T. Dai, and X. Gu, "A study on container vulnerability exploit detection," in *2019 ieee international conference on cloud engineering (IC2E)*. IEEE, 2019, pp. 121–127.

[20] T. Yarygina and C. Otterstad, "A game of microservices: Automated intrusion response," in *Distributed Applications and Interoperable Systems: 18th IFIP WG 6.1 International Conference, DAIS 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings 18*. Springer, 2018, pp. 169–177.

[21] V. V. Sarkale, P. Rad, and W. Lee, "Secure cloud container: Runtime behavior monitoring using most privileged container (mpc)," in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*. IEEE, 2017, pp. 351–356.

[22] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[23] L. Kuang and M. Zulkernine, "An intrusion-tolerant mechanism for intrusion detection systems," in *2008 Third International Conference on Availability, Reliability and Security*. IEEE, 2008, pp. 319–326.

[24] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: detecting code injection attacks with precision and efficiency," in *Proceedings of the 2013 ACM SIGSAC conference on computer & communications security*, 2013, pp. 1181–1192.

[25] "Sysdig Documentation," https://docs.sysdig.com/en/, [Online].

[26] M. Kerrisk, “Strace Documentation,” https://man7.org/linux/man-pages/man1/strace.1.html, [Online].
[27] “Seed lab documentation,” https://seedsecuritylabs.org/labs.html, [Online].
[28] seed-labs 2.0, “Cross-Site Request Forgery Attack Lab,” https://seedsecuritylabs.org/Labs_20.04/Web/Web_CSRF_Elgg/, [Online].
[29] ——, “Cross-Site Scripting Attack Lab (Elgg),” https://seedsecuritylabs.org/Labs_20.04/Web/Web_XSS_Elgg/, [Online].
[30] ——, “SQL Injection Attack Lab,” https://seedsecuritylabs.org/Labs_20.04/Web/Web_SQL_Injection/, [Online].
[31] E. Gerzi, “Compromising Kubernetes Cluster by Exploiting RBAC Permissions,” https://www.rsaconference.com/library/Presentation/USA/2020 /compromising-kubernetes-cluster-by-exploiting-weak-rbac-permissions, [Online].
[32] Kubernets.io, “Restrict a Container’s Syscalls with seccomp,” https://kubernetes.io/docs/tutorials/security/seccomp/, [Online].
[33] docker docs, “Seccomp security profiles for Docker,” https://docs.docker.com/engine/security/seccomp/, [Online].