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

Anonymous Author(s)

ABSTRACT

In recent years, cloud-native applications are widely hosted and managed through containerized environments due to their unique benefits, such as lightweight, portable, cost-efficient. In spite of their benefits and popularity, the security of such environments is very often questioned as evidenced by recent attacks. Among which, attackers are frequently using malicious code injection to breach systems and steal sensitive data from a containerized environment. However, there is no existing solution to efficiently detect malicious code injection in container applications. In this paper, we fill in this gap and propose a multi-level monitoring-based approach, where we monitor container activities at both system call level as well as orchestrator level. Thus, our approach can detect malicious code injection attacks in containers by distinguishing between their expected and unexpected behavior from various system call characteristics (e.g., sequence, frequency, etc.) along with activities through log files. We implement and evaluate our approach for Kubernetes, a major container orchestrator.

KEYWORDS

Container security, malicious code injection, Kubernetes, multi-level monitoring

ACM Reference Format:

Anonymous Author(s). 2018. On Detecting Malicious Code Injection by Monitoring Multi-level Container Activities. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The use of containers has substantially increased over the past several years as well as their popularity. According to a survey in 2022 by Cloud Native Computing Foundation (CNCF) [1], 44% companies are using containers for all of their products[1]. One of the most widely used container platforms, Docker [2] involves 13M+ developers, 7M+ applications, and 13B+ monthly image downloads. Additionally, one of the most popular container orchestrators (that assists in automatically deploying and managing large-scale container applicaitons), Kubernetes [9] is reported to be used in 83% of containerized production environments.

[1]https://www.cncf.io/reports/cncf-annual-survey-2022/

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Although the use of container technology is growing everyday, it also exposes us to new security threats [24]. For example, due to a flaw in the Azure Container Registry (ACR), personal files could be accessed by unauthorized users [25]. Also, security researchers reveal that Docker containers are often exploited for crypto mining by injecting malicious code into a container [15]. Furthermore, Docker Hub [2], a repository for Docker container images, is reported to be compromised when a number of benign images are infected by malware [14]. Data breach, resource depletion, malware propagation, and reputational harm are among the major consequences of these attacks against container technology, underscoring the necessity for security measures to detect unauthorized access and malware insertion.

As evident by those attacks, malicious code injection can be considered as one of the most dangerous risks to containerized environments [31], which gives attackers the opportunity to access private information, stop operations, or use the container as a launchpad for other attacks. By taking advantage of vulnerabilities in the application code, an attacker can introduce malicious code into an application that is executing inside a container. For instance, lack of input sanitization, an attacker may introduce harmful code into it (which we have simulated in this research work). To recognize and stop security risks that may jeopardize the integrity and confidentiality of applications running in containerized environments, it is important to work with malicious code injection in containers. But it is more difficult to detect an injection of malicious code in containerized environments than usual due to a number of specific characteristics of containers. For instance, because of container's dynamic nature (lightweight and flexible), it is difficult to monitor and detect malicious code injection. Additionally, because of the container's restricted visibility, it may be challenging to keep a watch for the injection of malicious code, especially if the attacker has access to the host system that the container is running on. Moreover, containers often have a newer attack surface, which means there are several possible entry points for attackers to insert malicious code, such as the container image, the container runtime, or the application operating within the container. Combination of all of these factors make it difficult for conventional detection systems to detect malicious code injection and respond appropriately.

The existing works (e.g., [8, 11, 12]) on detection of malicious code injection may face the following challenges:

- As previously mentioned, containers have a large attack surface. As a result, there are plenty of opportunities for an attacker to inject malicious code at different level of containers. So, it is necessary to monitor different levels of containers (Fig. 1), which is absent in existing works.
- To mitigate malicious code injection attacks, system call analysis may be a promising solution (as shown in [8, 11, 12]).

However, rather than concentrating on specialized solutions for certain attack pathways, the current solutions offer an generalized technique to prevent attacks on containers. As for these many attack pathways, applying a single generalized solution may not produce the desired results, rather a combination of diverse detection mechanisms that have not been used in existing works should be used to counter these threats.

- All system calls may not be equally important for analyzing anomalous behavior of containers created by malicious code injection. In fact, lots of system call can be identified as harmless [11]. If we filter out these unnecessary system calls, it may make our proposed mechanism more effective.

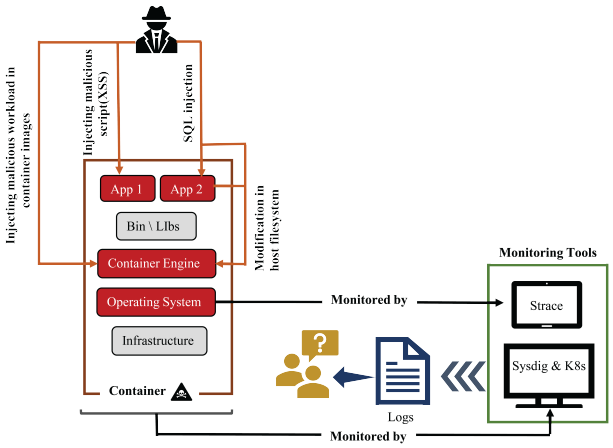


Figure 1: Proposed method to monitor different levels of container through different monitoring tools

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, instead of comparing various machine learning (ML) methods like most other existing works (e.g., [8, 12]) for a single detection mechanism, we alternatively concentrate on offering several detection mechanisms for various attack pathways at various container levels. Our rationale is to complement those existing ML-based approaches to be able to identify attacks carried out along alternative paths. While devising our solution, we use the attacks paths derived from many attack scenarios that are outlined by the Cloud Native Computing Foundation (CNCF) Finance User Group [22]. We employ the attack scenarios they provide since we find them helpful given that they are deployed in containers and take into account kubernetes as the orchestrator. Additionally, they cover a number of possible routes for injecting malicious code into containers. Our main contributions in this work are:

- We monitor container activities at both container orchestrator and system call levels (i.e., Figure 1) to detect malicious code injection attacks. As a result of multi-level monitoring, our solution can identify an attack in the following level even if an attacker manages to pass a single level.
- We derive the impacts of major 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, which covers numerous possible routes for a wide-range of malicious code injection attacks.
- We separate key group of system calls from all the system calls that show significantly more change during an attack. Frequency is employed to filter out useless system calls while detecting malicious activity. Thus, our solution becomes more efficient.
- We evaluate our solution for four different attack paths. Among all the attack paths provided by CNCF, these four paths finally lead to malicious code injection in container. Different number of containers (two to four) are used for simulating attacks on different paths. While working with frequency, we identify eighteen system calls out of twenty-eight as our key set of system calls based on thier observed changes during an attack. Our experiment results show improvements in performance metrics (accuracy, precision, recall, and f1 score) compared to [8], [11].

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. Finally, we conclude the paper in Section 6.

2 BACKGROUND AND LITERATURE REVIEW

This section first provides a background on the containerized technologies that will later be used in our approach and implementation. It then studies the existing works on addressing the possible solutions to different attacks in containerized environment and discusses their limitations.

2.1 Background on Container Technologies

Container. Multiple separate instances of a program can operate on a single host operating system thanks to containers, a type of operating system virtualization. This is accomplished by giving each container access to its own separated user space, file system, network stack, and process tree through the usage of the operating system's kernel. Since each container does not need its own operating system instance, containers are substantially lighter than virtual machines. Instead, they share the kernel of the host operating system, which lowers overhead and boosts efficiency. Images, which are portable, lightweight snapshots of a program, are the building blocks of containers.

Virtualization. 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 may save hardware costs, improve resource use, and increase scalability. Virtuallization techniques can be divided into two types [10]- 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 [32].

Kubernetes orchestrator. For the deployment, scaling, and maintenance of containerized applications, we need an orchestration tool. Kubernetes [3] is one of the most famous orchestration tools among them. Kubernetes is an open-source platform for container orchestration that streamlines the management, scaling, and deployment of containerized applications. 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. Our work uses a localized version of Kubernetes, namely, Minikube [4], because of its flexible usefulness as a monitoring tool.

System call. A system call (e.g., read, write) is a request made by a program operating in user space to use an operating system service. They give user space applications a controlled way to access resources and features that would otherwise be restricted. System calls play a significant part in managing interactions between the host operating system and the container in systems that use containers. Containers employ a virtualization technique known as "kernel namespace isolation" to make their own environment. As a result, a container's system calls are only visible to that container and are not visible to other containers or the host operating system. This maintains each container's isolation and security. Additionally, to limit the number of system calls that a container may make, container runtimes like Docker utilize a feature known as "seccomp" (secure computing mode) [21]. Seccomp restricts system calls to a limited set, therefore preventing attacks that take advantage of system call flaws.

Attack path. The CNCF Financial User Group [22] documents an in-depth threat modeling exercise performed against a generic Kubernetes cluster. Their work provides a detailed view of several attack paths by which an attacker can exploit configuration vulnerabilities within Kubernetes to achieve specific goals. CNCF identifies possible security problems at the platform's trust boundaries by examining the Kubernetes architecture. They also identify a few attack vectors (service token, compromised container, network endpoints, and RBAC concerns) from the trust boundaries, and by utilizing these, they create an attack tree with several branches that ultimately result in the injection of malicious code into containers. That attack tree's objective is to get a cluster run malicious code.

2.2 Existing works

On detection of attack through malicious code injection, several works are already in existence. We go through mostly those research works that focus on detecting malicious code injection in containerized environment. However, some of the works that do not involve containerized environment are also beneficial for our research purpose. The existing study on intrusion detection system at the container level is summarized as follows.

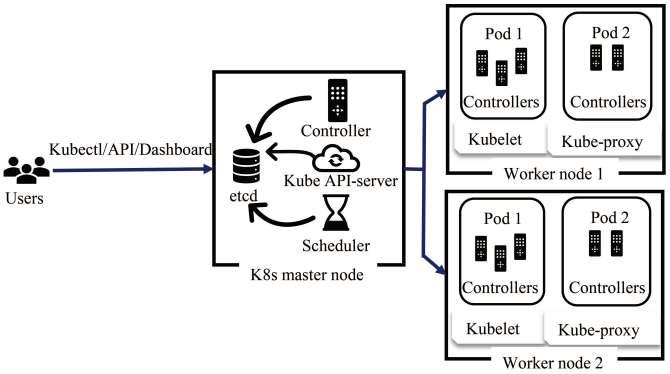


Figure 2: Structure and working principle of Kubernetes

For Linux containerized systems, Abed *et al.* [8] suggest a host-based intrusion detection solution. Cavalcanti, Inácio and Freire [12] propose a container-level anomaly based intrusion detection systems for multi-tenant applications using machine learning algorithms. In this literature, the researchers mainly work on four different vulnerabilities - Authentication Bypass, DoS Overflow, System User Privilege Escalation and Integer Overflow. Lin *et al.* [23] gather a dataset of attacks, including 223 vulnerabilities that work on the container platform, and use a two-dimensional attack taxonomy to categorize them. Next they use 88 common attacks fished out of the dataset to assess the security of the current Linux container technology. Chelladhurai *et al.* [13] address crucial Docker container security challenges as well as related work that is being done in this field. Also, they suggest security approaches and algorithms to deal with problems caused by DoS attacks in the Docker container technology. Tunde-Onadele *et al.* [33] propose detection mechanism using frequency of selective system calls that can detect more than 20 out of 28 tested vulnerability exploits. They only work with four specific system calls and the reasoning for choosing only these system calls is absent. The articles described above all work 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 those attack, all of them propose a common solution - mostly using sequence of system calls. Unlike our work, they do not propose an attack specific detection mechanism.

Yarygina and Otterstad [34] propose a flexible, cost-conscious intrusion response system for microservices that employs a game-theoretic method to automatically respond in real time to network threats. The research work of Thanh Bui [10] 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.* [31] discuss the possible security issues raised by the usage of containers and offers suggestions for resolving them. Sarkale, Rad and Lee [26] suggest 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 works provide 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. However, unlike us, none of them implements their suggested methodology and evaluated it using any kind of performance metrics, let alone work on a specific attack signature. As a result, the effectiveness of those works is difficult to confirm.

Castanhel *et al.* [11] and Hofmeyr and Somayaji [18] work on detecting anomaly using sequence of system calls and different machine learning algorithms. They provide how the change of length of system calls can impact intrusion detection using sequence of system calls. They do 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 be employed in containers. For network intrusion detection systems (NIDS) that use numerous independent components, Kuang and Zulkernine [20] suggest 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. Son et al. [30] design, implement and evaluate, DIGLOSSIA, a novel runtime tool for the accurate and efficient detection of code injection threats. However, this tool is implemented in a non-containerized environment unlike ours.

3 METHODOLOGY

From the perspective of an attacker, there are several ways to attack a container-based system, such as using virtualization to steal users' private data, initiating attacks by injecting malicious workload, or escalating the intrusion to numerous VM instances [11]. We follow the steps described in this section to cover each of these ways.

3.1 Approach Overview

Our work contributes to the mitigation of all the attack paths provided by CNCF that finally lead to malicious code execution in a container. To this end, we first deliberately perform some normal activities in our containers. Then we simulate several attacks, for instance, Cross Site Request Forgery (CSRF), Cross Site Scripting (XSS), SQL injection (SQLi) and Role Based Access Control attack (RBAC attack). For both benign and malicious activities we collect system call characteristics (e.g., sequence, frequency and name) and activity log file using a variety of tools (strace, sysdig and kubernetes dashboard) (Steps 1(a), 2(a) and 3(a) of Fig. 3). Secondly, we conduct system call pre-processing to filter out the minimal set of impactful system calls. Lastly, using both benign and malicious dataset, we build a model that can distinguish between benign and malicious behavior in a container. We elaborate on each of these steps as follows.

3.2 Modeling Benign Behavior

We purposefully visit several benign profiles to determine the typical frequency of benign activity, and then we use the data to determine the usual frequency (threshold) of each system call. Our dataset requires to be divided into a training set and a testing set even though our program do not employ any machine learning techniques. Both benign and malicious data are included in our training set. We identify the top k impactful system calls from the training set based on the frequency at which they change from benign to malicious activities. The lower bound of the frequency for each system call is determined using the average frequency of each system call from the benign component, and the higher bound is set using the average frequency of each system call from the malicious part. The threshold is established by averaging the upper and lower bounds. The summarized diagram is presented in Step 2(c) of Fig. 3. Next, we explain the block diagram presented in Step 3(c) of Fig. 3. We perform some innocuous profile editing and gather the standard sequence of system calls for them in order to obtain the standard sequence of system calls. In this context, we apply the Bag of System Calls (BoSC) and sliding window techniques [8]. This method extracts a sliding window from the entire list of system calls that can range in length from two to ten. The frequency of each system call in that order is then listed. The 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.

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 create a benchmark of system call parameters (frequency, sequence) by following the steps provided in this section. We then utilize this benchmark to subsequently identify the injection of malicious code into a container.

3.3 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 use different kinds of attack on container. For Fig. 4a, SQLi attack is implemented. XSS attack follows the path of Fig. 4b. For Fig. 4c and Fig. 4d, CSRF and RBAC attack are simulated respectively. Apart from the RBAC attack, the others are implemented using the attacks demonstrated in SEED-LABS 2.0 [6].

For four alternative attack pathways, we simulate four different types of attacks. We utilize a built-in website that functions like a social networking platform in SEED-LABS 2.0 [6] for two of them (XSS and CSRF). Certain users on this website have the ability to edit their profiles, view the profiles of others, make friend requests, etc. On the other hand, for SQLi, a company website with a database is deployed using two containers where employees can see their profiles and make changes to certain fields. In the following, we describe each attack type.

CSRF attack. The benign social networking website is combined with a malicious container that includes 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 [27].

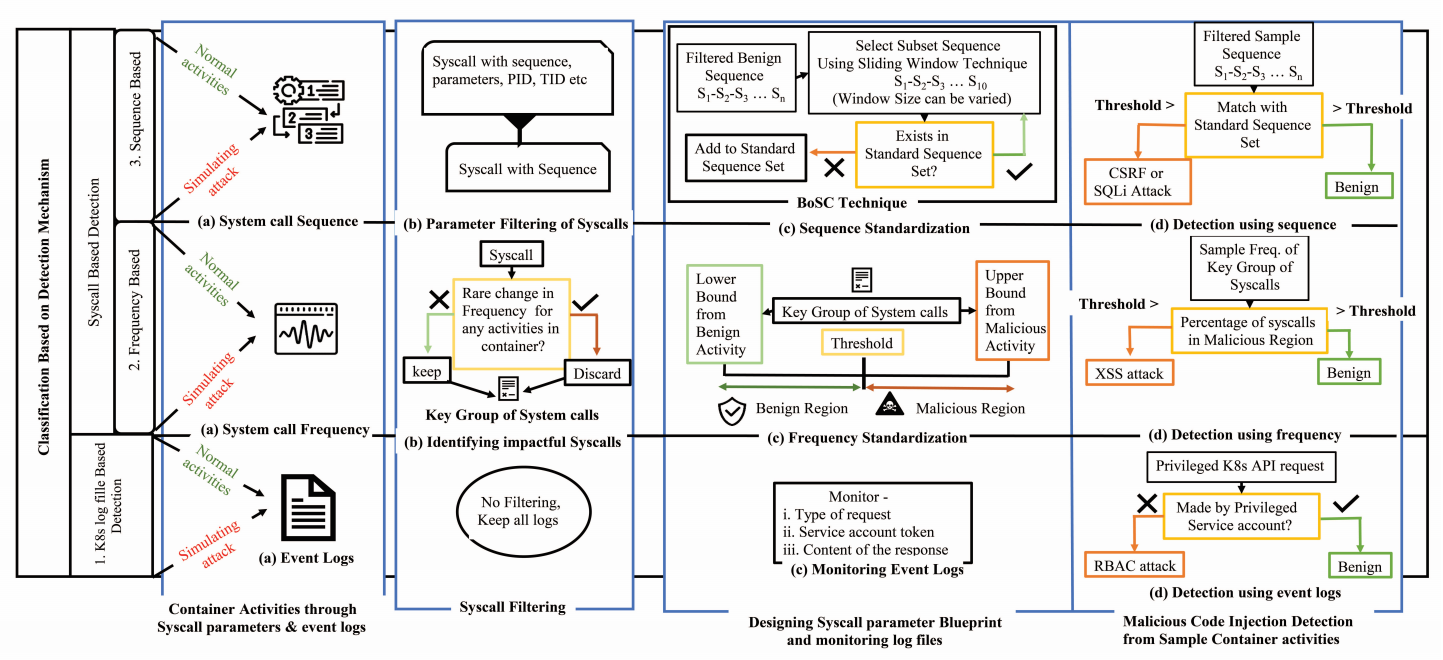


Figure 3: Overview of our proposed approach

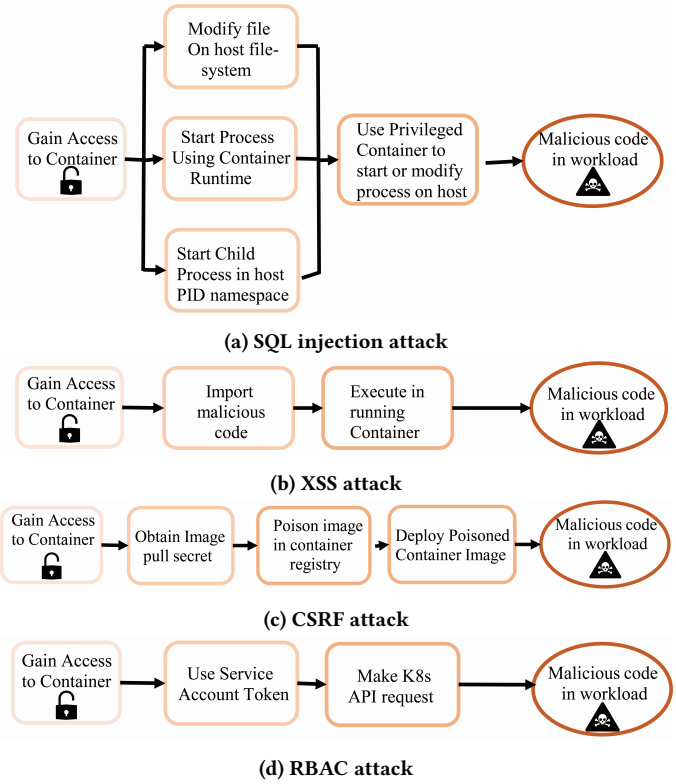


Figure 4: Attack paths of SQLi, XSS, CSRF and RBAC attacks

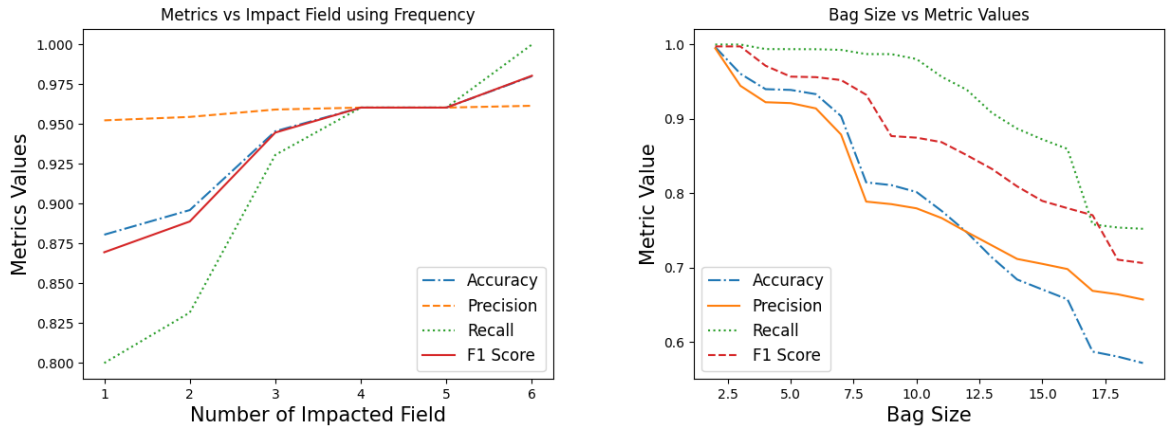
XSS attack. In this approach, one of the website's users (malicious) infect the benign website. On his/her profile area, s/he develops malicious workload that may influence other people's profiles. This attacker's profile is changed without his/her knowledge when a decent user visits it [28].

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 should already be familiar with the database table [29].

RBAC attack. A service account with the privilege (or role) of generating containers in the kube-system namespace is used to simulate a RABC or role-based access control attack against Kubernetes. Now that a container is 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 [17].

3.4 Segregation of Malicious Code Injection from Benign Activities

Prior to now, we establish 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 one to six, we are able to modify the attack's intensity. In each instance, we compare 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 use a trial-and-error methodology to get at this 82% so that we



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

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

Figure 5: Performance Metrics for two different proposed method

Values along X-axis in Fig. 6a and 6b		
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

Table 1: Definition of sets of system calls for Fig. 6a and 6b

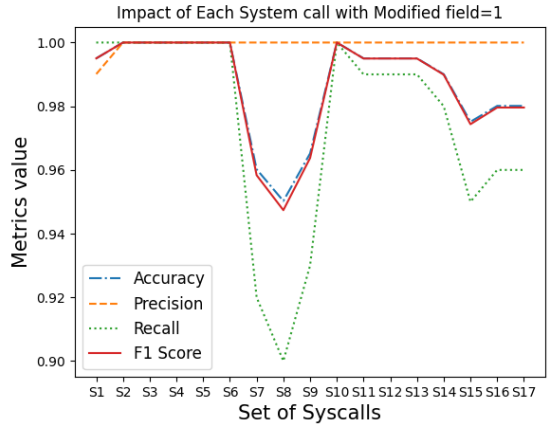
Values along X-axis in Fig. 6c and 6d		
Set	Modified Field=3	Modified Field=4
S1	fdatasync	pread
S2	S1+write	S1+fdatasync
S3	S2+pwrite	S2+write
S4	S3+fsync	S3+pwrite
S5	S4+io_submit	S4+fsync
S6	S5+sched_yield	S5+io_submit
S7	S6+futex	S6+sched_yield
S8	S7+nanosleep	S7+nanosleep
S9	S8+munmap	S8+futex
S10	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

Table 2: Definition of sets of system calls for Fig. 6c and 6d

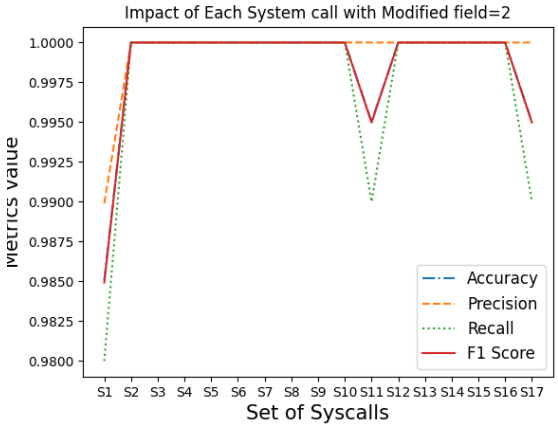
may have the best recall score. For our simulation, this 82% is the threshold mentioned in Step 2(d) of Fig. 3.

Here, we explain the Step 3(d) in Fig. 3. We already have a list of distinct sequences from the benign profile edit while dealing with the sequence of system calls. We create 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 are used. Following that, these sequences are contrasted with the normative sequences. We classify an activity as benign if more than 80% of the newly created sequence match the reference sequence; otherwise, the activity is labeled as malevolent behavior.

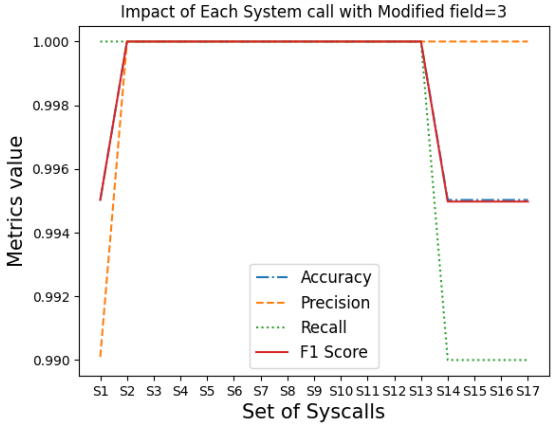
To mimic a SQLi attack, an attacker requires knowledge of the tables in a database hosted on the host server. s/he must have root privileges to do it. Thus, we must open a trace file on the database container and purposefully gain root access to it. Many unique system calls are created in this instance, and we compare them to the typical system calls made by a non-root user. In this instance, there is a presumption, though. We suppose that we retrain 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 is 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 [21] tool is already



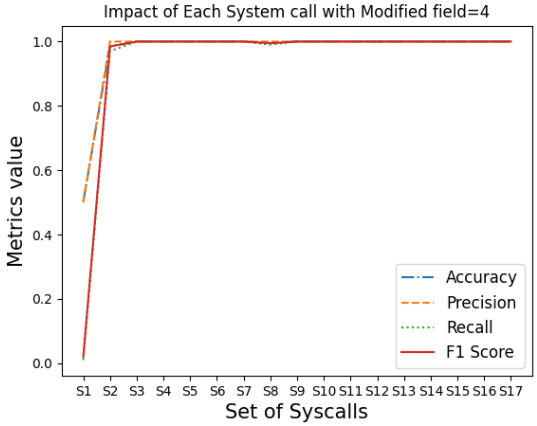
(a) Choosing the best set of system calls for attack severity=1



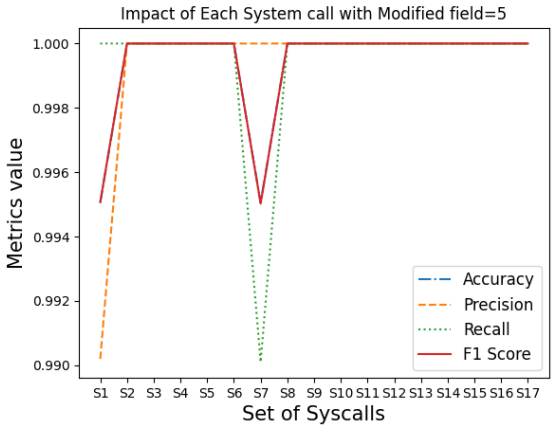
(b) Choosing the best set of system calls for attack severity=2



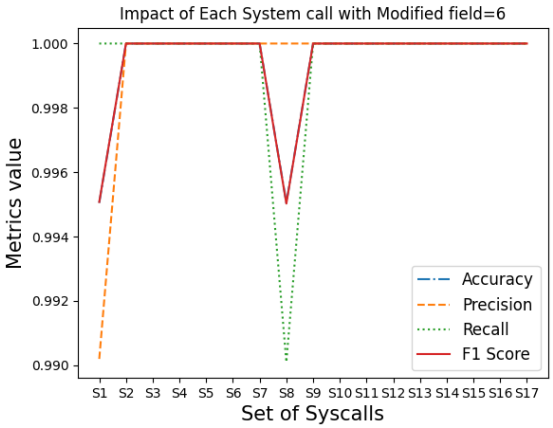
(c) Choosing the best set of system calls for attack severity=3



(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

Figure 6: Finding key group of system calls for different intensity of attack

Values along X-axis in Fig. 6e and 6f		
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

Table 3: Definition of sets of system calls for Fig. 6e and 6f

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 (e.g.,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. This refers to Step 1(d) of Fig. 3.

4 IMPLEMENTATION

This section details our implementation.

4.1 Implementation Setup

Our Docker containers are running on the host operating system seed-Ubuntu 20.04. Minikube v1.28.0 [4] is utilized to deploy our containers in Kubernetes. With the help of Minikube, we can show Kubernetes on a local computer. We have to generate our own dataset for each of the attacks because there is no dataset that contains the system calls made a container following our attack pathways.

4.2 Dataset Generation

Using the Sysdig tool [7], the frequency of each system call is 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 have only recorded the frequencies immediately following a simulated attack. We have utilized the strace tool [19] 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 have launched strace tool on a target process and downloaded each and every system call made throughout the course of the attack. The outcome is kept in a log file.

Minikube Dashboard is 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 is made to mimic an RBAC attack in the kube-system namespace. As a result, we have used the log files from kube-apiserver, one of the pods in the kube-system namespace, to gather data. Seccomp tool [21] [16] 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.

4.3 Pre-processing data

We use Python v3.10.1 to pre-process the data. We gather frequency data for both benign and malicious actions to process the frequency data. Despite the fact that throughout an action, several unique system calls are produced, not all of them are equally affected by an attack. Hence, we only consider system calls that display a notable deviation from customary behavior. We discover that around twenty out of all system calls exhibited this behavioral modification.

We also filter out any extraneous data gleaned from the strace and sysdig tools, which is applied to both frequency and order of system calls. For instance, system calls' arguments, time-stamps, and other details are removed.

5 EXPERIMENTS & RESULTS

This section presents our experimental results.

5.1 Detection using Frequency

We change the malicious workload size (from infecting one field to six fields) in our experiment to reflect the intensity of each attack in our situation. As the intensity of attack increases, the performance of our detection mechanism increases as well (as shown later). Fig. 5a shows that performance metrics (accuracy, precision, recall, and f1 score) become greater in value as the number of infected fields grows (i.e., as the attack severity increases). We restrict our attack severity at infecting six fields since our detection method behaves consistently as attack severity increases. Fig. 5a is not observed using the entire set of system calls we have collected while monitoring our container. We have 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. We have simulated our suggested approach using these top k system calls since they show a substantial shift as the attack intensity is altered. The impact of each system call is then assessed by gradually adding them to the initial set. Six graphs are produced for various workload lengths. The values along X-axis in Fig. 6a and 6b are explained in Table 1, Fig. 6c and 6d are explained in Table 2, and Fig. 6e and 6f are explained in Table 3.

We may infer from Fig. 6a that up to the set S6, we are 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 ten to twelve most significant system calls from the table. We have mostly chosen these top ten to twelve system calls, which have demonstrated regular frequency, to build the graph in Fig. 5a.

5.2 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 [18]. Fig. 5 shows the result of the performance metrics of detection using sequence.

5.3 Detection using event logs

We continuously keep monitoring the condition of each pod in master node with the help of minikube dashboard. In our case, as our RBAC attack tree suggests that a kubernetes API request is made to the master node, we check each event logs of the master node pod called kube-apiserver [5] in kube-system namespace. During this monitoring time, our main focus is to check - the type of API request made to the pod, the service account token that makes the API request to the pod and the content of the response of this pod against the API request. Judging by these three factors, it should be evident that whether this API request is a privileged one or not. If this appears to be a privileged one but the service account token involved with this API request belongs to the default namespace, then we identify this service account token as a suspicious one.

6 CONCLUSION AND FUTURE WORK

For detecting malicious code injection in containers, we provided various detection approaches in this study for various attack vectors. Our suggested method was comprised of 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 had divided the top system calls that exhibit notable variations in response to container activities.

There exist limitations in our work. First, currently, we focus solely on the attack vectors that ultimately resulted in the insertion of malicious code. In addition to them, there can be more ways to inject malicious code in containers. Detection using sequence of system call has another disadvantage over frequency detection that it might take longer to identify malicious activity if the sequence is too lengthy. We leveraged Kubernetes dashboard to spot RBAC attacks. We manually kept a track of the log files, which need to be automated in future. Also, the mechanism presently chooses the settings on the basis of trial-and-error method which can be misleading if generated dataset is small. 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.cncf.io/. [Online, last accessed: December 24, 2022].
[2] https://hub.docker.com/. [Online, last accessed: November 23, 2022].
[3] . https://www.g2.com/categories/container-orchestration. [Online, last accessed: May 13, 2023].
[4] https://minikube.sigs.k8s.io/docs/start/. [Online, last accessed: May 15, 2023].
[5] https://kubernetes.io/docs/concepts/overview/components/. [Online, last accessed: May 14, 2023].
[6] Seed lab documentation. https://seedsecuritylabs.org/labs.html. [Online, last accessed: May 2, 2022].
[7] Sysdig Documentation. https://docs.sysdig.com/en/. [Online, last accessed: June 15, 2022].
[8] 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)*, pages 1–5. IEEE, 2015.
[9] aqua. Kubernetes in Production: What You Should Know. https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-in-production-what-you-should-know/. [Online, last accessed: May 14, 2023].
[10] T. Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.
[11] 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)*, pages 1–6. IEEE, 2021.
[12] 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*, pages 1–9, 2021.
[13] 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)*, pages 856–859. IEEE, 2016.
[14] 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, last accessed: February 17, 2023].
[15] 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, last accessed: February 17, 2023].
[16] Docker. Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/. [Online, last accessed: October 11, 2022].
[17] 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, last accessed: May 6, 2022].
[18] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180, 1998.
[19] M. Kerrisk. Strace Documentation. https://man7.org/linux/man-pages/man1/strace.1.html. [Online, last accessed: May 28, 2022].
[20] L. Kuang and M. Zulkernine. An intrusion-tolerant mechanism for intrusion detection systems. In *2008 Third International Conference on Availability, Reliability and Security*, pages 319–326. IEEE, 2008.
[21] Kubernets.io. Restrict a Container's Syscalls with seccomp. https://kubernetes.io/docs/tutorials/security/seccomp/. [Online, last accessed: August 10, 2022].
[22] M. Lancini. The Current State of Kubernetes Threat Modelling. https://blog.marcolancini.it/2020/blog-kubernetes-threat-modelling/, June 30, 2020. [Online, accessed May 16, 2023].
[23] 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*, pages 418–429, 2018.
[24] J. N. Müller. Static source code analysis of container manifests.
[25] K. Rajapakse. Azure container registry image scanning. https://faun.pub/azure-container-registry-image-scanning-aaeae84d1c0c, May 18, 2020. [Online, last accessed: July 08, 2022].
[26] 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)*, pages 351–356. IEEE, 2017.
[27] seed-labs 2.0. Cross-Site Request Forgery Attack Lab. https://seedsecuritylabs.org/Labs_20.04/Web/Web_CSRF_Elgg/. [Online, last accessed: May 3, 2022].
[28] seed-labs 2.0. Cross-Site Scripting Attack Lab (Elgg). https://seedsecuritylabs.org/Labs_20.04/Web/Web_XSS_Elgg/. [Online, last accessed: May 4, 2022].
[29] seed-labs 2.0. SQL Injection Attack Lab. https://seedsecuritylabs.org/Labs_20.04/Web/Web_SQL_Injection/. [Online, last accessed: May 3, 2022].
[30] 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, pages 1181–1192, 2013.

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

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

[33] 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)*, pages 121–127. IEEE, 2019.

[34] 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, Madrid, Spain, June 18-21, 2018, Proceedings 18*, pages 169–177. Springer, 2018.