2020-06-04

Rule-Based Security Monitoring of Containerized Environments

Gantikow, H

http://hdl.handle.net/10026.1/15847

10.1007/978-3-030-49432-2_4
Springer International Publishing

Rule-based Security Monitoring of Containerized Environments

Holger Gantikow[1], Christoph Reich[2], Martin Knahl[3], and Nathan Clarke[4]

[1] science+computing ag, Atos, Tübingen, DE
gantikow@gmail.com
[2] Institute for Cloud Computing and IT Security, Furtwangen University, Furtwangen, DE
christoph.reich@hs-furtwangen.de
[3] Faculty of Business Information Systems, Furtwangen University, Furtwangen, DE
martin.knahl@hs-furtwangen.de
[4] Center for Security, Communications and Network Research, Plymouth University, Plymouth, UK
N.Clarke@plymouth.ac.uk

Abstract. Containers have to be secured in a multi-tenant environment. To secure the use of containerized environments, the effectiveness of a rule-based security monitoring approach have been investigated.
The approach of this paper can be used to detect a wide range of potentially malicious behaviour of workloads in containerized environments. Additionally is able to monitor the actual container runtime for misuse and misconfiguration. In order to evaluate the detection capabilities of the open-source tools utilized in a container, various scenarios of undesired behaviour are closely examined. In addition, the performance overhead and functional limitations associated with workload monitoring are discussed. The proposed approach is effective in many of the scenarios examined and its performance overhead is adequate, if appropriate event filtering is applied.

Keywords: Container Virtualization · Docker · Security · Monitoring · Anomalous Behaviour · System Call Tracing

1 INTRODUCTION

Virtualization at the operating system level using containers has gained popularity over the past few years, mainly driven by the success of Docker. As all containers share the same kernel of the underlying Linux host system, a lower resource overhead compared to virtual machines can be achieved with this lightweight virtualization type (Felter et al., 2015). This is particularly important when deploying an application and its dependencies independently of the underlying host system is at the center of interest, as it is important with micro-service architectures, for example. Containers are often used to provide basic components, such as web servers, databases, service discovery services or message brokers. In addition, containers offer the advantage that they can be easily integrated into suitable *Continuous Integration, Delivery and Deployment* (CI/CD) tools and pipelines and can be distributed to be run on different runtimes with the help of several standards as specified by the *Open Container Initiative* (OCI) (Open Container Initiative, 2017a; Open Container Initiative, 2017b). Containers have also made their way into the domain of *High Performance Computing* (HPC). Containerized HPC environment have the benefits of deploying user-provided code, improving collaboration through simplified distribution, allow simple reproducibility, and have nevertheless a low performance overhead. For containers used in HPC, a number of solutions were developed, which differ significantly in terms of isolation mechanisms, that are the focus of this paper. These HPC specific solutions include *Shifter* (Jacobsen and Canon, 2015), *Charliecloud* (Priedhorsky et al., 2017), and above all *Singularity* (Kurtzer et al., 2017).

Recent surveys (Sysdig, 2018) show, that Docker deployments are still most widespread, with a share of 83% of the investigated systems. For this reason, the authors of this paper focuse on the Docker container runtime, although, many aspects can be applied to other solutions as well. However the development of general purpose container runtimes is ongoing and alternatives to Docker are appearing, such as Podman (Containers Organization, 2019). Approaches that focus strongly on security, but are characterized by a much smaller deployment rate, are for example SCONE (Arnautov et al., 2016) and gVisor (Young et al., 2019). Besides of the great efforts of increasing the security of the technology, in particular Docker (Combe et al., 2016), is sometimes viewed critically. Most frequently it is listed as one of the major challenges (Portworx, 2018), when deploying container technology in production. It should be noted that the situation has improved significantly in recent years with the addition of complementary security options. However, these are often disabled by default or being deactivated to ensure smooth operation in terms of compatibility with a wide range of applications. However a current survey indicates that 94% of the participants still have container security concerns and fear a rise of container security incidents (Tripwire, 2019).

To further increase the security level of containerized environments, this paper propose applying rule-based security monitoring to containerized environments. This paper [5] explores the suitability of the approach for detecting a) various types of undesired behaviour that might indicate misuse and attacks of workloads running *inside a container*, and b) misconfigurations and attempts to extend privileges and reduce isolation mechanisms in place at the *container runtime level*.

The rest of this paper is organized as follows: *Section 2* provides a brief introduction to security mechanisms that can be applied to containerized environments. *Section 3* describes the special monitoring characteristics regarding containerized workloads. *Section 4* discusses related work concerning container virtualization. *Section 5* introduces the rule-based security monitoring approach evaluated in *Section 6*. *Section 7* discusses limitations of the proposed approach and future work. The paper concludes in *Section 8*.

[5] **Note:** This paper is both a revised and extended version of a previous publication (Gantikow et al., 2019). The present version is characterized by an extension of the security monitoring approach beyond the containerized workloads to the container runtime itself, in order to recognize misconfigurations and attempts to weaken isolation settings or extend privileges there.

2 CONTAINERS AND SECURITY

The Linux Kernel features *Control Groups* (cgroups) and *Namespaces* are used to provide resource limiting (CPU, memory, IO) and process isolation and represent the basic components of containers. These mechanisms are used to protect the host and other containers from resource starvation by one container and to provide containerized processes a confined instance of the underlying global system resources. Linux currently provides the seven namespaces *cgroups, IPC, Network, Mount, PID, User, UTS*.

Meanwhile, most container runtimes provide support for already established Linux security mechanisms that complement the essential mechanisms *cgroups* and *namespaces*. To provide *Mandatory Access Control* (MAC) *AppArmor* and *SELinux* can be utilized, which use the *Linux Security Module* (LSM) framework. These technologies provide means to limit the privileges of a process and thus mitigate harmful effects in the event of an attack.

Another Linux Kernel feature to decrease risks arising from undesired behaviour is the *Secure Computing Mode* (seccomp), which makes it possible to implement a very basic sandbox in which only a reduced number of system calls are available. In that way, individual unneeded system calls can be specifically denied. However, this requires a high degree of knowledge of the system calls required in a specific containerized workload and must therefore be adapted on a per-container basis. The default seccomp profile provided with Docker only disables 44 of over 300 system calls (Docker Inc., 2019) to maintain wide application compatibility while providing a somewhat higher level of protection. This includes sane defaults that can be applied to all containers, such as blocking the *reboot* system call, which denies that a reboot of the host system can be triggered from inside a container.

There exist further *preventive security measures* to secure containerized environments, described in greater detail in an overview paper (Gantikow et al., 2016),

These includes at the Linux Kernel level the possibility to remap the container root user inside a container to a non-privileged user outside the container (*User Namespaces*) or to use the *Linux capabilities*, which divide the privileges of the superuser into distinct units, and can be activated or deactivated individually. However, capabilities are inferior to seccomp in terms of granularity. For example, the capability *SYS_ADMIN* bundles a very large set of functionalities, which could be used to deactivate further security measures (Walsh, 2016).

Another important role play tools that perform static analysis for vulnerabilities on container images (*CVE scanner*), since they can be used to prevent images with vulnerable code from being available or launched in the containerized environment in the first place.

In general, the isolation provided by containers is still to considered to be weaker than that of hypervisor-based approaches. While for instance *Denial of Service (DoS)* protection provided by cgroups is to be considered effective as long as appropriate limits are defined, better protection against *Information Leakage* is still in development.

Currently not all Linux subsystems are namespace-aware and in many locations more information about the host system can be collected than it would be possible in a virtual machine. Above all the */proc* file system should be mentioned here, because it offers many possibilities for information leakage (Gao et al., 2017), which can provide information for a tailor-made attack.

The risk of *privilege escalation*, allowing the modification of files on the host system or a container breakout with full privileges on the host system, pose a serious problem as well. This is often caused by misconfiguration, for example when containers are started with elevated privileges or reduced isolation. For example, starting a container with the *–privileged* flag causes the container to behave like a process with elevated rights outside of a container as all capabilities are granted to the container. This flag is required for special use cases, such as running Docker inside Docker and should never be used in a regular scenario (Stoler, 2019).

Therefore a complementing preventive security measures is proposed adopting a *rule-based security monitoring*, with the aim to detect misuse and common attacks in containerized workloads, as well as misconfigurations and attempts to weaken isolation or escalate privileges at the runtime level.

For the scope of this paper, we consider some widespread attack scenarios and take advantage of containerization specific characteristics for monitoring. Use cases at the containerized workload level include unauthorized file access as it can precede information leakage, unexpected network connections and application startup, and attempted privilege escalation. Use cases monitored at the runtime level include exposure of the Docker REST API, various container lifecycle events and modifications to security and performance isolation settings.

3 CONTAINERS AND MONITORING

The monitoring of containerized workloads requires a new approach, since traditional agent-based approaches cannot be applied directly. Docker Containers in particular follow the concept of application containers, which are made up of exactly one process per container. This is in contrast to the operating system containers used by LXC, which, while using the shared host kernel for all containers, start them as almost full-fledged Linux systems that behave almost like a VM and allow for several services. Adding a monitoring agent to a container would break the *single process per container* model and moreover require a modification of the container image. The need for image modification can be an unacceptable condition in environments where users demand the integrity of the images they provide. Therefore, adding a process to a container that performs the monitoring functionality *from within a container* is usually not advisable.

Also less suitable is the approach of using *sidecar containers*, which are started additionally and take over the monitoring function in a dedicated way. These would have to be started with elevated privileges and would result in increased resource consumption if a dedicated sidecar container were started for each container.

Therefore, this approach rely on one monitoring agent per host. Thus no image adjustment has to be performed, the one process per container model can be maintained and no additional overhead is caused by sidecar containers. In addition, a feature of container virtualization is beneficial: containerized workloads are transparent to the host system as regular processes. This means that a much more accurate view of the virtualized workload can be collected by the host system with containers than would be possible with VMs. Also, the collected state information can be directly assigned to the corresponding workload since there are no concurrent activities in a container that could distort the picture, as likely caused by the guest system in a VM.

As a data source, our approach relies on the system calls that are issued by the monitored containers. These get enriched with further context by the tools presented in Section 5. System calls have the advantage that they map exactly what a process is currently performing. In principle, further traditional performance metrics such as CPU load, memory usage, I/O throughput could be included in the system described for additional insights, but is not implemented as of now.

4 RELATED WORK

There are several projects available to increase the security in containerized environments, that build up on top of the basic container security technologies briefly introduced in Section 2. The use of the complementing Linux Kernel security features Capabilities, Seccomp and MAC is being recommended by a measurement study on Container Security (Lin et al., 2018). According to the authors, these mechanisms provide more effective measures in preventing privilege escalation than the basic isolation mechanisms (i.e. Namespaces and Cgroups) that build the foundation of containers.

The projects relying on the complementing Linux Kernel security features include policy generators such as *LicShield* (Mattetti et al., 2015), which generates container-specific AppArmor profiles based on a learning phase - or *SPEAKER* (Lei et al., 2017), which divides the time a container is running into different phases and assigns optimized custom seccomp profiles to them. This benefits from the fact that a service usually requires significantly fewer unique system calls after the start and initialization phase.

The exclusive use of performance metrics collected at the hypervisor level for security monitoring is discussed in approaches like (Nikolai, 2014). His effort to detect malicious behaviour in hypervisor-based environments can be transferred to containerized environments, but does not provide the accuracy system calls can provide. Furthermore, an approach based on performance data only offers the possibility to determine *what* is happening, but not *why*, unless additional data sources can be analyzed.

The usage of system calls for detecting malicious behaviour dates back to the seminal work of Forrest. She proposed the usage of system call sequences to distinguish between normal and anomalous behaviour (Forrest et al., 1996). Concurrent processes such as background processes did however affect the detection accuracy. This situation is improved by the use of containerized workloads, since the one application per container approach significantly reduces concurrency that leads to distortion of results.

Abed (Abed et al., 2015) investigates an approach where traces collected by *strace* are used for anomaly detection without prior knowledge of the containerized application. However, mimicry attacks (Kang et al., 2005) can be used to circumvent the utilized Bag of System Calls approach.

Borhani (Alex Borhani, 2017) provides a paper that reports on the real-world feasibility of Falco, but focuses less on its limitations but on the *Incidence Response* aspect.

The combination of system calls to detect anomalies with different approaches from the field of Machine Learning has been investigated by a number of authors. Among them Maggi (Maggi et al., 2010), who used Markov models for system calls and their arguments and (Koucham et al., 2015) who extended them with system call specific context information and domain knowledge. An approach with neural networks based on convolutional and recurrent network layers is offered by Kolosnjaji (Kolosnjaji et al., 2016). His approach increased the detection rate of malware. Focus on distributed data collection and processing of large amounts of data is put by Dymshits (Dymshits et al., 2017), who uses an LSTM-based architecture and sequences of system call count vectors.

To the best of our knowledge, we are not aware of literature that addresses security monitoring of the container runtime interfaces.

5 OVERVIEW OF PROPOSED SYSTEM

Our approach is based on the Open Source tools *Sysdig* (Sysdig, 2019b) and *Falco* (Sysdig, 2019a), which will be introduced in the following briefly. They differ from other options, such as *strace* and *eBPF* (Fleming, 2017), by native support for several container engines, including Docker, LXC and rkt. This support enables filtering of collected data based on individual containers, specific system calls, file name patterns, or network connection endpoints. The ability to use filters also significantly reduces collection and processing effort.

5.1 Sysdig

Sysdig uses two core components to implement its functionality. A kernel module (*sysdig_probe*) that uses the Linux Kernel facility *tracepoints* serves as a collection component to capture all system calls of a process (or containerized process) as *events*. These traces are then passed to a daemon that serves as the processing component.

Sysdig combines the functionality of a number of well-known analysis tools, including *strace*, *tcpdump* and *lsof* and combines them with transaction tracing. The combination of the individual functionality offers a considerably deeper view of the system and individual processes than would be possible with a single tool. Therefore, Sysdig is also well suited for error analysis on a system. As already mentioned, Sysdig offers various possibilities for filtering to reduce the amount of data collected. In the context we utilized it, filtering for individual system calls, arguments passed to them, such as file names, source of an event, like the container or process name,were beneficial.

5.2 Falco

Falco is best described as a *behavioural activity monitor* that complements Sysdig's system call capture functionality with the ability to detect anomalous activities based on rules. To do so it relies on the same *sysdig_probe* kernel module for system call capturing.

At the core of the applicable rules are Sysdig filter options, which are referred to as *conditions*. If a condition is evaluated as *true*, meaning that an event matched the given requirements, the event is flagged as anomalous behaviour. This presupposes, however, that exact knowledge of the desired behaviour of a container is available to be able to define the conditions that specify a violation. An anomalous activity could include starting an unauthorized process, accessing unusual paths in file systems, outgoing network connections, or attempts to modify system binaries and configuration files.

However, Falco can only detect point anomalies, i.e. single events where the described conditions are met, such as the usage of an undesired system call. The rule set currently does not provide means to allow the detection of collective anomalies or contextual anomalies where multiple conditions or additional preconditions have to be met. This limits the detection capabilities to a certain extent, as the critical examination in the evaluation will show. In addition, Falco only *detects* behavioural anomalies. A subsequent mitigation beyond logging and notification derived therefrom is currently not provided.

5.3 Architecture

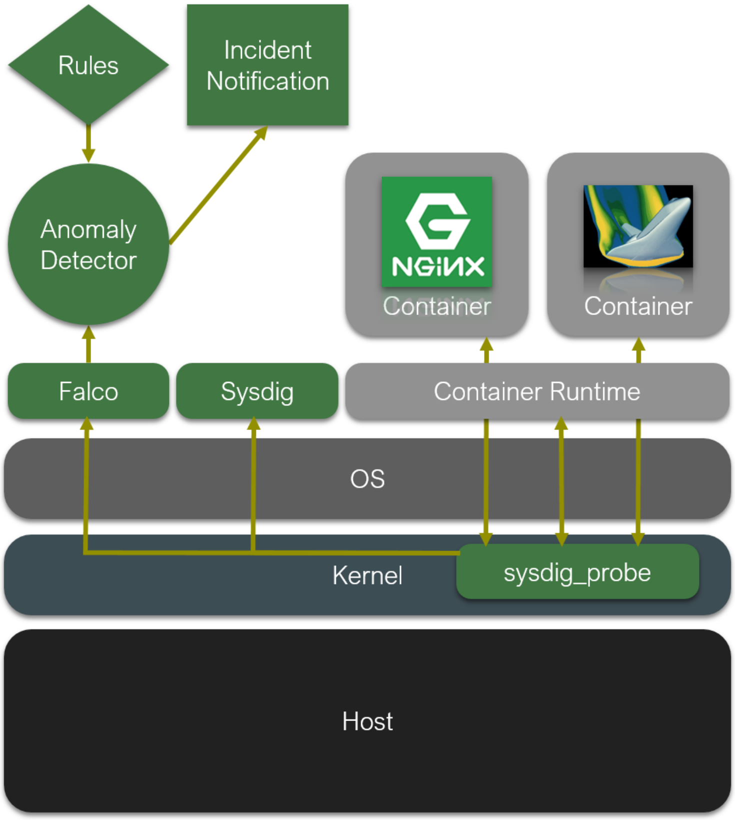

Fig. 1. Proposed Rule-based Security Monitoring Architecture, based on (Gantikow et al., 2019).

The architecture (see Fig. 1) is based on the corresponding figure from (Gantikow et al., 2019), but has been adapted to reflect the extension of the rule-based monitoring approach to the container runtime.

As already touched, both Sysdig and Falco use the same (*sysdig_probe*) kernel module for capturing system calls. System calls are the means by which user-level processes interact with the operating system when they request services provided by it, such as opening a file or initiating a network connection. As system calls are the only entry point into the kernel to request such functionality, they are of high value when it comes to capturing the behaviour of a process. The Sysdig kernel module not only captures the identifiers of the system calls, but also the call parameters used and the return values received, which provides further possibilities for evaluation.

Sysdig is responsible for the display and processing of the recorded events. It was primarily used to create and test the Sysdig filters to be used as Falco conditions for classifying an event as anomalous behaviour. The actual classification is carried out by Falco. Falco uses rules to be created in advance (Figure 1, *Rules*) to decide on deviant behaviour and can perform a basic notification (Figure 1, *Incident Notification*) using logging frameworks, plain e-mail or messengers after a rule matches.

We are currently investigating how the current *Incident Notification* can be extended to *Incident Mitigation* based on the severity of an event. One feasible and minimalistic approach would be to pause the workload, which is characterized by undesired behaviour, retrieve the full state for further analysis and to restore it after analysis if appropriate.

6 EVALUATION

In order to examine the applicability of the rule-based approach, security monitoring was applied to two relevant layers in a containerized environment: a) *Container Execution Level* (Section 6.2), which concerns misuse and attacks occurring inside of containerized workloads, and b) *Container Runtime Level* (Section 6.3), takes misconfigurations and attempts to weaken isolation or extend privileges at the container runtime interface itself into consideration.

The investigation on the two layers is each divided into two phases. In the initial phase it was investigated if and how Sysdig can be used to detect the behaviour regarded as harmful. If the behaviour was detectable using a Sysdig filter, a Falco rule was derived from this filter. In the second phase, the rule was then tested for its suitability for automated detection of undesired behaviour.

We would like to point out that Sections 6.1 *Test Environment* and 6.2 *Studied Misuse and Attacks at Container Execution Level* were incorporated unmodified from the paper (Gantikow et al., 2019), since both the test environment and the evaluated use cases at container execution level remained unchanged for this extended version. The same applies to Section 7.1 *Performance Evaluation*, as the section has remained unmodified as well.

6.1 Test Environment

The evaluation was carried out in a virtualized test environment. This means, referring to Figure 1, that the host system is actually a virtual machine in our case. However, this has no influence on the functionality of the described approach. We used the following components in a VM with 1 Core and 4GB of memory and did not impose additional container resource limits, unless otherwise noted:

OS: Debian GNU/Linux 9.5 (stretch)
Kernel: 4.9.0-8-amd64
Docker: Docker version 18.06.1-ce, build e68fc7a
Sysdig: 0.24.1
Falco: 0.13.0

Unless otherwise specified Debian GNU/Linux 9.5 (stretch) as container image has been used.

6.2 Studied Misuse and Attacks at Container Execution Level

In order to examine the rule-based security monitoring approach for its suitability to detect common misuses and attacks as they may occur in containerized workloads, a series of scenarios was defined and evaluated for their detectability. In the following section, these scenarios are briefly presented and the creation of the Sysdig filter and the Falco rule are presented on an exemplary basis.

Unauthorized File Access
For the test setup of the *Unauthorized File Access* we used the deliberately insecure web application *WebGoat* (OWASP, 2018). We used the already-available *webgoat* image on Dockerhub and evaluated the detectability of the task *Bypass a Path Based Access Control Scheme*. This represents a *directory traversal* attack where the successful attacker can access files outside the root directory of the web server. Such an attack is often used to gain access to configuration files with passwords.

The following code represents the Sysdig *condition*, which also serves as the basis for a Falco rule set.

```
container.name=webgoat
and evt.type=open
and evt.dir="<"
and fd.type=file
and not (fd.directory contains "/webapp")
```

The condition is true if the container name [6] is *webgoat* and the system call *open* accesses a file outside a path containing the string */webapp*.

[6] The *container name* refers to the name of the running container instance as returned by *docker ps* and not to the image name in general. The respective container ID (*container.id*) could also be used as an alternative to the name.

The example shows that it can be easily generalized and can be used, in addition to restricting a service to its corresponding root directory, to monitor activity to *access to non-namespaced resources* or other resources that could lead to information leakage. An attempted write access, for example to directories containing system binaries, can also be detected in this way. Furthermore, the condition itself can be extended by further event parameters that have to be fulfilled, so that it is possible to detect when a process not approved for this purpose tries to access a specific device.

Start of Unauthorized Application

A similarly well generalizable test case is the detection of the *Start of Unauthorized Applications* inside a container.

```
list: authorized_processes
 items: [ps, hostname]
condition:
 container.name=debian-test
 and evt.type=execve
 and evt.dir="<"
 and not (proc.name in (authorized_processes))
```

The condition thus recognizes the execution of programs that are not *ps* or *hostname*. This can be used to allow a container to start only its corresponding service and to log, for example, if a crypto miner or a (remote) shell is started, as might be the case in a successful remote attack. The list *authorized_processes* serves as a white list here.

Container Breakout (using *nsenter*)

Another test case examined was the detection of certain processes that are related to specific threats, i.e. are maintained on a black list if necessary. In this example the command *nsenter* was used to run a process within the name spaces of another process, which is detectable by filtering for the system call *setns*. Although this mechanism is typically blocked from within a container by other measures, there is still a risk of misconfiguration. In addition, it can be useful to be able to log the access from the host into a container by this procedure by adapting the container identifier to *container.id = host*.

Unexpected Network Connection

In order to detect if a container establishes undesired connections to the internet, for example to download malicious code for an exploit or to open a remote shell, the detectability of *Unexpected Network Connections* was also examined. This can be implemented by creating a white list with approved targets or limiting it to specific TCP ports.

Loading of Kernel Module

Although it is not possible in the default configuration to load kernel modules on the host from the container, the recent breakout from a Docker evaluation environment

Play with Docker (Stoler, 2019) inspired us to consider this case. As described in the referenced case, this can lead to a privilege escalation with full administrative privileges on the host and thus control over additional containers.

Denial of Service (DoS)

Even though (if applied) cgroups can prevent a resource starvation of the host and other containers in terms of CPU shares and memory, there is the possibility, depending on the configuration, to fill up shared file systems, which is why this test case had to be investigated.

Buffer Overflow

The last test case examined was whether it is in principle possible to detect buffer overflows using our rule-based approach. Abusing a buffer overflow is a common security exploit, so that memory areas with executable code are overwritten with malicious code, which can be the basis for an attack.

6.3 Security Monitoring at Container Runtime Level

Inspired by attacks made possible by misconfigurations that expose the Docker REST API to the Internet (Chikvashvili, 2019), the previous work of monitoring containerized workloads was extended to also monitor the container runtime / engine for unwanted behaviour. This approach is also in line with the recommendations of NIST, which recommends to examine the container runtime more closely for risks (Souppaya et al., 2017). The advantage of monitoring on this layer is that these activities are independent of the workload in the containers, i.e. no prior knowledge is required. Thus the rules can be activated directly on a large number of hosts in a containerized environment.

We chose three areas of priority for the monitoring of the container runtime: a) Monitoring the access to the *Docker REST API*, with the focus of detecting misconfigurations that expose the API over network. Monitoring the Docker *command-line interface* (CLI) (*docker* * commands) for b) *container lifecycle events* and c) the modification of *container security settings*. Scenarios b) and c) are particularly relevant for environments in which users, e.g. members of the docker group, have direct access to the *docker* CLI without *role-based access control* (RBAC) mechanisms or restrictive wrappers around the CLI being in place. In particular, c) can be misused in various ways to extend privileges on the host to a level similar to *root* access and reduce security and performance isolation. We have identified a number of misuse scenarios that are briefly presented below.

Docker REST API

Since Docker 0.5.2 the REST API endpoint, which is used by the Docker CLI to interact with the Docker Engine, switched from a (locally bound) TCP socket to a UNIX socket, where access is controlled by the traditional UNIX permissions. The configuration option to globally expose the REST API over HTTP(S) can still be enabled. As this may lead to an attacker taking over the Docker Host, the hosts that are able to connect should be restricted to trusted hosts only and the API endpoints should be secured with HTTPS and certificates. We identified the following undesired events:

Exposure of API Since exposure of the Docker REST API endpoint over the network can have high security implications, monitoring for such misconfigurations should be performed. The indicator used herefore is a process binding to port TCP/2375 (HTTP) or TCP/2376 (HTTPS).

Connection to API In addition to monitoring for exposure, it should also be monitored whether connections are established to these ports. This is important, if the monitoring was started after the exposure of the API endpoints.

Container Lifecycle Events

The Container Lifecycle Events offer a variety of options that are well-suited to be monitored. We have grouped the commands according to their functionality and their potential for misuse, which we will describe briefly below.

Typical Lifecycle Events Commands in this group represent typical lifecycle commands (*pull, run, exec, pause, unpause, start, stop, kill*), which for instance are used to download images or start a container. These commands usually have no security-critical context, but should be logged for auditing reasons. Especially if *exec* is used to execute commands inside an already running container, as this indicates administrative intervention.

Circumvention of Registries On a production system, it is unlikely if an image is not downloaded from an image registry that might contain curated images, but is loaded locally from the file system. In such a situation, this could mean that an attacker is trying to launch an image that he has previously manually downloaded.

Publication of Images The same level of unlikelihood applies to saving, exporting or uploading an image. Such events appear out of place on a production system and could indicate that an attacker tries to export container images and extract them.

Exposure of Services In an environment where containers provide network accessible services, publishing a service from the container is a valid intent. In environments where this is not the case, the event when container ports are exposed to the host could indicate misuse, such as to provide access for third parties.

Container Security Settings

Unrestricted access to the Docker CLI provides a variety of controls that may allow a user to run a container with privileges very close to an administrative account on the host system. Alternatively, restrictions can be modified to weaken the default isolation of a container. We have identified a number of attack scenarios that should be monitored accordingly.

Usage of Additional Host Resources It is possible to pass host resources into the container. These can be local file system directories, complete devices, including GPUs. While there are use cases where this can be a practical approach, such an event can have far-reaching security implications, as this can provide privileged access to the resource for the user starting a container this way.

Weakening of DoS Protection When starting a container, a large number of performance limits can be imposed, such as limits for CPU and memory consumption. There are also additional settings that may be used to protect against fork bombs (PID limit) or the *Out of Memory* (OOM) killer. According to (Chelladhurai et al., 2016), the activation of these limits is a suitable mean to ward off the effects of an attempted DoS through a container. Consequently, modifications to these settings must be viewed critically. In the default setting, no resource limits are active. However, in production operation such limits should be applied.

Modification of Namespaces An adjustment of the namespace isolation can lead to reduction of the default isolation, thus reduced security. Especially if a namespace is set to *=host*, as this results in a sharing of the corresponding host namespace with the container. Rare occasions where this would be acceptable include monitoring containers that require an unrestricted system view.

Extension of Privileges The strongest possibility to extend the privileges through a container is the option of starting a container in *privileged* mode. This results in the container receiving the full set of capabilities and also cgroups-based limitations not being applied. This results in a containerized process that has at least the same privileges as if run directly on host. As the security implications are severe (see (Stoler, 2019)) this behaviour is only justified in very rare cases, such as requiring *Docker in Docker* or as a *temporary* workaround.

Modification of Security Options The *–security-opt* setting is one of the most powerful controls as it allows to carry out adjustments on a variety of security tools that were integrated later, This flag is used to modify SELinux settings, specify the AppArmor profile to be applied, grant the gaining of new privileges and also specify which seccomp profile to use or disable seccomp protection altogether. Modifications made here should therefore be examined with particular caution.

Modification of Performance Restrictions In addition to the DoS protection settings, there are other settings to distribute the host's capacity between multiple containers. These include measures to limit the IO throughput of devices. Manual adjustments should be examined critically.

6.4 Results

Table 1 summarizes the detectability of the scenarios presented in Section 6.2 regarding *Misuse and Attacks at Container Execution Level*. For each scenario it is shown whether a filter for general detection can be created with Sysdig, whether a automated detection

Table 1. Summary of the detectability of various misuse and attack scenarios using Sysdig and Falco, based on (Gantikow et al., 2019).

Scenario	**Sysdig**	**Falco**	**Indicator**
Unauthorized File Access	Yes	Yes	Violation of white list with authorized files and directories
Start of Unauthorized Applications	Yes	Yes	Violation of white list with authorized application names
Container Breakout	Yes	Yes	Black list - *nsenter* called - or violation of white list
Unexpected Network Connection	Yes	Yes	Violation of white list with authorized communication partners
Loading of Kernel Module	Yes	Yes	Black list - *insmod* called - or violation of white list
Denial of Service	Yes	No	Frequency of occurrences
Buffer Overflow	No	No	Not applicable

can be implemented with Falco, and what can be used as an indicator of the undesired behaviour.

As the table shows, it is possible to create a rule for Falco in almost all of the cases investigated, if the event can be detected by a Sysdig filter. Many of the examined cases can be restricted in terms of examined objects through the use of a white list or black list based approach. This is possible for example with the detection of non-authorized file access, the start of a non-authorized application - or similar, easily derivable scenarios. However, this requires an exact knowledge of the workload to be examined and an adjustment of the lists on a per image basis.

Since Falco does not currently provide support for the occurrence frequency of an event, it is currently not possible to use Falco for the detection of DoS attacks. One-time access to a service itself can be captured by Sysdig - and thus also converted into a Falco condition. However, the frequency of events cannot be taken into account.

The recognition of a buffer overflow is also not feasible as intended, since it can normally also not be recognized by static analysis. The execution of a particular exploit that can trigger a buffer overflow could in principle be detected by relying on the system calls and their respective order. However, Falco can only detect anomalies based on a single system call, not on their sequence in a particular order. In addition, the approach would not be generalizable and remain tied to a specific exploit. Blocking the execution of an exploit using the process identifier or binary name is not worth the effort, as this can be circumvented by renaming it. The exclusive operation of approved applications and the detection of unapproved network lines is more effective here.

Tables 2, 3, 4 present the results of the evaluation of *Security Monitoring at container runtime level*, broken down into the three areas, as of Section 6.3. The tables list in each case the title of the scenario, the feasibility of detection with Sysdig and Falco, as well as the respective indicators for the occurrence of the threat.

The results show a distinct picture. In all three areas it is possible to detect the corresponding events through Sysdig filters with very little effort and to convert them into corresponding Falco rules, which allow for an automated monitoring. In most cases it is sufficient to filter the call of the Docker CLI for the corresponding sub command or command line parameters. The search terms used are specified in the column of the respective scenario as *indicator*.

The proposed approach for monitoring on the container runtime layer offers the possibility to capture a wide range of potentially malicious behaviour with limited effort. Even if the lifecycle commands are rather logged for audit purposes, the method is useful to get an overview of events in the environment, especially if users are granted interactive access to the Docker CLI. As mentioned before, it is possible to roll out identical rule sets to monitor the runtime layer to a large number of container hosts in a containerized environment without having to know about the workloads running inside the containers. However, one should think about how to classify the severity of each event [7], since pausing a container is considered much less critical than starting a privileged container.

Table 2. Summary of the detectability of exposure of and connections to the *Docker REST API* using Sysdig and Falco.

Scenario	**Sysdig**	**Falco**	**Indicator**
Exposure of API	Yes	Yes	`bind()` to TCP/2375 or TCP/2376
Connection to API	Yes	Yes	Connection to TCP/2375 or TCP/2376

Table 3. Summary of the detectability of various *Docker Lifecycle Events* using Sysdig and Falco.

Scenario	**Sysdig**	**Falco**	**Indicator**
Typical Lifecycle Events	Yes	Yes	`pull, run, exec, pause, unpause, start, stop, kill`
Circumvention of Registries	Yes	Yes	`import, load`
Publication of Images	Yes	Yes	`export, save, push`
Exposure of Services	Yes	Yes	`--publish , -p, --publish-all , -P`

7 DISCUSSION

Our research has shown that there are a number of limitations in the tools used that should not go unmentioned. For example, Falco does not support DoS detection because the rules used do not support the frequency of occurrence of an event. Here one would

[7] Currently Falco provides the priority categories *emergency, alert, critical, error, warning, notice, informational, debug*.

Table 4. Summary of the detectability of modifications to *Docker Security Settings* using Sysdig and Falco.

Scenario	**Sysdig**	**Falco**	**Indicator**
Usage of Additional Host Resources	Yes	Yes	`--mount, -v, --volume, --device, --gpu`
Weakening of DoS Protection	Yes	Yes	`-m, --memory, -c, --cpu, --blkio-, --device-, --pids-limit, --ulimit --oom-kill-disable`
Modification of Namespaces	Yes	Yes	`--ipc, --network, --pid, --userns, --uts`
Extension of Privileges	Yes	Yes	`--privileged mode, --cap-add`
Modification of Security Options	Yes	Yes	`--security-opt`
Modification of Performance Restrictions	Yes	Yes	`*iops, *bps`

wish for a threshold value to be adjustable in the rules, so that notifications take place only if an event occurred n times during a specific time interval.

Falco's rules also do not allow the integration of load sensors, so that for example a notification could be given when certain load thresholds have been reached, or these could be used as a decision-making assistance for alleged false positives.

It has been shown that profound knowledge about the properties and requirements of a containerized application is almost mandatory for the creation of rules. Although there are certainly generalizable rules, the example of the web application (see Section 6.2) shows that applications regularly have a different locations where they keep their data. Other individual properties include the name of the service to be run in the container, so that all other applications can be blocked, or which network connections are necessary. Depending on the environment, this may not be a concern, for example, if an environment consists of a large number of containers, all of which are started from the same image. In this case, the appropriate rule set can be applied to all of the running containers and be maintained on based on the shared image. In more individual environments, or environments in which users may use containers for interactive work, the configuration effort is significantly higher.

The white/black list approach also has limitations, i.e. when an attacker knows the content of the lists and can prepare himself accordingly. Especially file name-based approaches can be bypassed by renaming files easily.

However, the tools used are characterized by the fact that they can also be applied to other container runtimes. Although we have only tested the use with Docker, rules for monitoring containers can be applied to containers started with other runtimes. For the monitoring of the Runtime CLI a higher adaptation effort is necessary, as the rules utilized where Docker CLI specific.

Table 5. Sysdig overhead for various statistics of sysbench-fileio benchmark in comparison to baseline run without Sysdig.

sysbench-fileio statistic	**Sysdig with full capture**	**Sysdig with filter**
Operations performed (total)	1,81%	0,00%
Requests/sec executed	4,72%	2,97%
Total number of events	4,72%	2,97%
Total time taken by event execution	5,48%	0,67%
Per-request statistics: avg in ms	10,53%	3,51%
Average Overhead	5,45%	2,02%

7.1 Performance Evaluation

To determine the performance overhead caused by security monitoring, we made use of a traditional benchmark tool: *Sysbench* (Kopytov, 2019), of which we created a containerized version. To exclude buffering effects when using the filesystem-level benchmark *fileio* included with Sysbench, we used it with a test file four times the amount of the available memory. During each 5min run of the benchmark a corresponding capture file was created with Sysdig and afterwards several performance indicators of the *sysbench-fileio* run were evaluated.

In order to be able to rate how high the benefit of using filters is a) a *full capture* and b) a capture with an active *filter* was created, which limited the recording to the *open()* system call, as one would use if one only wanted to log file accesses of a container. As baseline served measurements of the *sysbench-fileio* without activated Sysdig capturing and all benchmark runs (deactivated Sysdig, Sysdig with full capture, Sysdig with filter) were averaged over three runs each. All runs were performed in the same virtual test environment described in Section 6.1.

It was observed that over several measurements the average overhead in case a) (full capture) was 5,45%, whereas the use of the filter reduced the overhead in case b) to 2,02%. The use of the filter also affected the size of the capture file. In b) only 270 events needed to be recorded, resulting in a 1,2MB trace file, whereas the unfiltered case a) logged 3.270.522 events in a 270MB file on average. This implies, that if possible, filters should be activated for data and overhead reduction. The overhead, broken down by individual *sysbench-fileio statistic*, is shown in Table 5.

8 CONCLUSIONS

The investigated approach has shown the general applicability of a rule-based approach for monitoring containerized environments. The focus was on the monitoring of workloads running in containers and of the interfaces of the Docker Container Runtime. It has been shown that the approach can detect a variety of undesired behaviour with a low performance overhead. In addition, the creation of an appropriate set of rules, especially for the monitoring of commands sent via the CLI, can be done with moderate effort.

However, when monitoring containerized workloads, automated rule creation should be performed, since the requirements typically differ from workload to workload, i.e. in most cases they can only be generalized on a per-image basis. In cases where it is not possible to create a corresponding set of rules, one should consider the use of a behaviour monitor that compares the current behaviour against a stored reference model. However, this approach would also require a certain amount of time for the creation of a behaviour model, so that this approach cannot be applied directly as well.

Scenarios that are not yet covered should also be considered, e.g. if commands such as *nsenter* or the associated system call *setns*) can be used to execute commands within a running container by bypassing *docker exec* since its use is monitored.

It is planned that future work will address security monitoring of distributed workloads, where shared workloads strongly interact across host boundaries. Sysdig and Falco already offer corresponding interfaces that cover container schedulers like Kubernetes. We are also interested in further automating rule generation and introducing incident mitigation beyond notification.

Bibliography

[Abed et al., 2015]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. *2015 IEEE Globecom Workshops, GC Wkshps 2015 - Proceedings*.

[Alex Borhani, 2017]Alex Borhani (2017). Anomaly Detection, Alerting, and Incident Response for Containers. *SANS Institute InfoSec Reading Room*, (GIAC GCIH Gold Certification).

[Arnautov et al., 2016]Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Pietzuch, P., and Fetzer, C. (2016). SCONE: Secure Linux Containers with Intel SGX. *Osdi*, pages 689–704.

[Chelladhurai et al., 2016]Chelladhurai, J., Chelliah, P. R., and Kumar, S. A. (2016). Securing docker containers from Denial of Service (DoS) attacks. *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, pages 856–859.

[Chikvashvili, 2019]Chikvashvili, Y. (2019). Cryptocurrency Miners Abusing Containers: Anatomy of an (Attempted) Attack. [ONLINE] Available at: `https://blog.aquasec.com/cryptocurrency-miners-abusing-containers-anatomy-of-an-attempted-attack`. [Accessed 31 July 2019].

[Combe et al., 2016]Combe, T., Martin, A., and Di Pietro, R. (2016). To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing*, 3(5):54–62.

[Containers Organization, 2019]Containers Organization (2019). Podman. [ONLINE] Available at: `https://podman.io/`. [Accessed 31 July 2019].

[Docker Inc., 2019]Docker Inc. (2019). Seccomp security profiles for Docker. [ONLINE] Available at: `https://docs.docker.com/engine/security/seccomp/`. [Accessed 31 July 2019].

[Dymshits et al., 2017]Dymshits, M., Myara, B., and Tolpin, D. (2017). Process monitoring on sequences of system call count vectors. *Proceedings - International Carnahan Conference on Security Technology*, 2017-October:1–5.

[Felter et al., 2015]Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172.

[Fleming, 2017]Fleming, M. (2017). A thorough introduction to ebpf. [ONLINE] Available at: `https://lwn.net/Articles/740157/`. [Accessed 14 January 2019].

[Forrest et al., 1996]Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T. (1996). A sense of self for Unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 120–128.

[Gantikow et al., 2016]Gantikow, H., Reich, C., Knahl, M., and Clarke, N. (2016). Providing security in container-based HPC runtime environments. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9945 LNCS:685–695.

[Gantikow et al., 2019]Gantikow, H., Reich, C., Knahl, M., and Clarke, N. (2019). Rule-based Security Monitoring of Containerized Workloads. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, pages 543–550, Heraklion, Crete - Greece.

[Gao et al., 2017]Gao, X., Gu, Z., Kayaalp, M., Pendarakis, D., and Wang, H. (2017). ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. *Proceedings - 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*, pages 237–248.

[Jacobsen and Canon, 2015]Jacobsen, D. M. and Canon, R. S. (2015). Contain This, Unleashing Docker for HPC. *Cray User Group 2015*, page 14.

[Kang et al., 2005]Kang, D.-k., Fuller, D., and Honavar, V. (2005). Learning Classifiers for Misuse Detection Using a Bag of System Calls Representation. *Proceedings of the 2005 IEEE Workshop on Information Assurance and Security United States Military Academy, West Point, NY*, pages 511–516.

[Kolosnjaji et al., 2016]Kolosnjaji, B., Zarras, A., Webster, G., and Eckert, C. (2016). Deep learning for classification of malware system call sequences. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9992 LNAI, pages 137–149.

[Kopytov, 2019]Kopytov, A. (2019). Sysbench: Scriptable database and system performance benchmark. [ONLINE] Available at: `https://github.com/akopytov/sysbench`. [Accessed 14 January 2019].

[Koucham et al., 2015]Koucham, O., Rachidi, T., and Assem, N. (2015). Host intrusion detection using system call argument-based clustering combined with Bayesian classification. *IntelliSys 2015 - Proceedings of 2015 SAI Intelligent Systems Conference*, pages 1010–1016.

[Kurtzer et al., 2017]Kurtzer, G. M., Sochat, V., Bauer, M. W., Favre, T., Capota, M., and Chakravarty, M. (2017). Singularity: Scientific containers for mobility of compute. *Plos One*, 12(5):e0177459.

[Lei et al., 2017]Lei, L., Sun, J., Sun, K., Shenefiel, C., Ma, R., Wang, Y., and Li, Q. (2017). SPEAKER: Split-phase execution of application containers. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10327 LNCS, pages 230–251.

[Lin et al., 2018]Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., and Zhou, Q. (2018). A Measurement Study on Linux Container Security. In *2018 Annual Computer Security Applications Conference (ACSAC '18)*, pages 418–429, San Juan, PR, USA. ACM, New York, NY, USA.

[Maggi et al., 2010]Maggi, F., Matteucci, M., and Zanero, S. (2010). Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395.

[Mattetti et al., 2015]Mattetti, M., Shulman-Peleg, A., Allouche, Y., Corradi, A., Dolev, S., and Foschini, L. (2015). Securing the infrastructure and the workloads of linux containers. *2015 IEEE Conference on Communications and NetworkSecurity, CNS 2015*, (Spc):559–567.

[Nikolai, 2014]Nikolai, J. (2014). Hypervisor-based cloud intrusion detection system. *2014 International Conference on Computing, Networking and Communications (ICNC)*.

[Open Container Initiative, 2017a]Open Container Initiative (2017a). OCI Image Format Specification v.1.0.0. Technical report.

[Open Container Initiative, 2017b]Open Container Initiative (2017b). OCI Runtime Specification v.1.0.0. Technical report.

[OWASP, 2018]OWASP (2018). Owasp webgoat project. [ONLINE] Available at: `https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`. [Accessed 14 January 2019].

[Portworx, 2018]Portworx (2018). 2018 Container Adoption Survey. Technical report.

[Priedhorsky et al., 2017]Priedhorsky, R., Randles, T. C., and Randles, T. (2017). Charliecloud: Unprivileged containers for user-defined software stacks in HPC. *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, 17:p1–10.

[Souppaya et al., 2017]Souppaya, M., Morello, J., and Scarfone, K. (2017). Application container security guide. *NIST Special Publication 800-190*.

[Stoler, 2019]Stoler, N. (2019). How i hacked play-with-docker and remotely ran code on the host. [ONLINE] Available at: `https://www.cyberark.com/threat-research-blog/how-i-hacked-play-with-docker-and-remotely-ran-code-on-the-host/`. [Accessed 14 January 2019].

[Sysdig, 2018]Sysdig (2018). Docker Usage Report 2018 - An inside look at shifting container usage trends.

[Sysdig, 2019a]Sysdig (2019a). Sysdig falco: Behavioral activity monitoring with container support. [ONLINE] Available at: https://github.com/draios/oss-falco. [Accessed 14 January 2019].

[Sysdig, 2019b]Sysdig (2019b). Sysdig: Linux system exploration and troubleshooting tool with first class support for containers. [ONLINE] Available at: https://github.com/draios/sysdig. [Accessed 14 January 2019].

[Tripwire, 2019]Tripwire (2019). State of Container Security Report. Technical Report January.

[Walsh, 2016]Walsh, D. (2016). Container tidbits: Adding capabilities to a container. [ONLINE] Available at: https://rhelblog.redhat.com/2016/11/30/container-tidbits-adding-capabilities-to-a-container/. [Accessed 10 January 2019].

[Young et al., 2019]Young, E. G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2019). The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, page 16, Berkeley, CA, USA. USENIX Association.