2019

Rule-based Security Monitoring of Containerized Workloads

Gantikow, H

10.5220/0007770005430550
Proceedings of the 9th International Conference on Cloud Computing and Services Science
SCITEPRESS - Science and Technology Publications

Rule-based Security Monitoring of Containerized Workloads

Holger Gantikow[1], Christoph Reich[2], Martin Knahl[3] and Nathan Clarke[4]
[1]*Science+Computing ag, Atos, Tuebingen, Germany*
[2]*Institute for Cloud Computing and IT Security, Furtwangen University, Furtwangen, DE, Germany*
[3]*Faculty of Business Information Systems, Furtwangen University, Furtwangen, DE, Germany*
[4]*Center for Security, Communications and Network Research, Plymouth University, Plymouth, U.K.*

Keywords: Container Virtualization, Docker, Security, Monitoring, Anomalous Behavior, System Call Tracing.

Abstract: In order to further support the secure operation of containerized environments and to extend already established security measures, we propose a rule-based security monitoring, which can be used for the detection of a variety of misuse and attacks. The capabilities of the open-source tools used to monitor containers are closely examined and the possibility of detecting undesired behavior is evaluated on the basis of various scenarios. Further, the limits of the approach taken and the associated performance overhead will be discussed. The results show that the proposed approach is effective in many scenarios and comes at a low performance overhead cost.

1 INTRODUCTION

Over the last few years, Linux containers have led to a strong trend towards lightweight virtualization, where only the application and its dependencies represent the content of the virtualized unit, omitting a virtual machine with an independent operating system. Because the virtualization takes place at the operating system level, all containers running on one host share the very same Linux kernel, which in turn handles the virtualization. This leads to reduced resource requirements (Felter et al., 2015), that are particularly beneficial for scalable, distributed microservice architectures. For this purpose, containers are preferably used for underlying basic components such as databases, message brokers, service discovery services or web/application servers. In addition, container technologies play an important role when it comes to managing software in DevOps environments. There they offer, integrated into corresponding *Continuous Integration, Delivery and Deployment* (CI/CD) tools and pipelines, the possibility to package applications and their dependencies automatically in a standardized image format and to subsequently deploy them independent of the underlying Linux distribution. Containers have also established themselves in the domain of *High Performance Computing* (HPC) and Scientific Computing in general. There they serve with low performance overhead as appropriate means to improve the portability of applications across different systems and institutions, to simplify the handling of user-provided applications and to ensure the reproducibility of experiments.

According to current surveys (Sysdig, 2018), the most widespread container runtime is still *Docker* with a share of 83% of the investigated systems, followed by *rkt* with 12%. However, there is also a considerable number of runtime engines designed for a specific purpose or domain. Especially in the field of HPC, where *Charliecloud* (Priedhorsky et al., 2017), *Shifter* (Jacobsen and Canon, 2015) and especially *Singularity* (Kurtzer et al., 2017) are far more widespread than Docker, but in most cases use other isolation mechanisms than the general purpose runtimes, which are in the focus of this paper. While overall interest in utilizing containers due to their benefits is still rising, *container security*, or the lack of thereof (Combe et al., 2016), is still listed the most frequent challenge to overcome when adopting containers (Portworx, 2018), even though the situation has improved significantly in recent years as a result of the integration of several security mechanisms.

In this paper, we propose the use of rule-based security monitoring to increase the security level in a containerized environment. We explore the applicability of the approach to detect several types of misuse and potential attacks and discuss the limitations of the described approach, as well as the associated

Gantikow, H., Reich, C., Knahl, M. and Clarke, N.
Rule-based Security Monitoring of Containerized Workloads.
DOI: 10.5220/0007770005430550
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 543-550
ISBN: 978-989-758-365-0

monitoring-related performance overhead.

The rest of this paper is organized as follows. *Section 2* introduces the basic security mechanisms that exist in the domain of container virtualization. *Section 3* details which special characteristics exist regarding monitoring compared to hypervisor-based virtualization. *Section 4* discusses related work to further secure containerized workloads. *Section 5* presents the proposed rule-based security monitoring approach that is evaluated in *Section 6*. *Section 7* discusses the current limitations and future work, before we conclude in *Section 8*.

2 CONTAINERS AND SECURITY

Container isolate workloads by using Linux features like the *Control Groups* (cgroups) for metering and limiting resources as CPU share, memory and IO, as well as Kernel Namespaces. These provide processes their own limited view of the shared host system. Linux currently implements seven namespaces and is able to provide isolated instances of cgroups, IPC, Network, Mount, PID, User, UTS.

In addition to these fundamental isolation mechanisms, support for additional already established security mechanisms has been integrated into the container runtimes over time. These include, for instance, support for components such as *AppArmor* and *SELinux* that use the Linux Security Module interface and are used for implementing *Mandatory Access Control* (MAC). They can be applied to directly control the access rights of individual processes and thus reduce the risk of potentially harmful effects in the event of an error or attack. The kernel feature *Secure Computing Mode* (seccomp) allows to explicitly deny certain system calls of the Linux kernel for a container and to create a basic sandbox. For example, by disallowing the *chown* system call it can be prevented that inside a container the file ownership can be changed. The use of these mechanisms require in-depth knowledge of the requirements of a specific container. To maintain high compatibility with the widest range of containers the default profiles supplied typically contain a less restrictive set of rules and should be adjusted subsequently on a per container basis.

Complementing these essential mechanisms, further *preventive security measures* that can be applied to secure containerized environments are described in an overview paper (Gantikow et al., 2016). Among the most important are *CVE scanners* for the static analysis of vulnerabilities in container images, *user namespaces* to remap the root user inside the container to a less-privileged user on the host and *capabilities*, which allow superuser privileges to be dropped in functional groups. Seccomp is more fine-granular, as it is based on single system calls, whereas the capability *SYS_ADMIN* offers a very extensive range of functions, which in principle can be exploited to deactivate other security measures (Walsh, 2016).

Although there are many ways to ensure the operational security of containerized environments and to protect hosts and containers running in parallel from malicious workloads in one container, the isolation provided by containers is still considered to be weaker compared to hypervisor-based virtualization.

However, better protection against *Information Leakage* is still evolving, as the Linux kernel does not yet provide a fully developed approach to partitioning all available resources and subsystems that can be utilized by containers. There are still a number of limitations that make it possible to collect more information about the host from inside the container than would be possible from a VM. Especially the */proc* file system is problematic, because it exposes many information leakage channels (Gao et al., 2017), allowing an optimized attack by the information retrieved. Another potential area of threat is the risk of privilege escalation, i.e. the obtaining of elevated access rights to the system or even a container breakout. Even if the greatest risk here is posed by improperly running containers with elevated privileges (Stoler, 2019), for example by applying the *–privileged* flag at container startup, leading to security settings so that the container behaves practically like a process running outside the container, this risk should not be underestimated.

We therefore propose to add a security measure, the adoption of a *rule-based security monitoring of containerized workloads* with the aim of detecting misuse and attempted common attacks. For the scope of this paper, we focus on the detection of some widespread scenarios and take advantage of container-specific characteristics in monitoring. Use cases include monitoring for unauthorized file access as it can precede information leakage, launching unexpected network connections and applications, attempting privilege escalation, and monitoring a number of additional elements on the host system.

3 CONTAINERS AND MONITORING

While containers improve the flexibility and portability of workload execution, they make it difficult to

monitor these workloads when using traditional process and performance monitoring tools. Among the reasons for that is that it is considered best practice to isolate containerized workloads by separating them with a *single process per container* model. Adding a monitoring agent to each container would both break this model, thus the simplicity of containers, but also require the modification of the containerized image. The addition of such a third party application to user provided images may represent an intolerable condition for scenarios where users want to ensure the integrity of their code. Therefore the establishment of a monitoring process *within* a container is not suitable in most cases. The limited suitability also applies to the approach of a separate *sidecar container*, which is placed alongside each container and takes over the monitoring functionality in a dedicated manner. This approach would need extended privileges and suffers from high additional overhead.

Therefore, we suggest the use of one monitoring agent per host for our approach. This offers the advantages that the overhead caused by additional monitoring containers is eliminated, that the one process per container model can be preserved, that no monitoring agent has to be introduced into the individual containers and thus a modification of user supplied images can be omitted. By monitoring the system calls issued by a container, the host is able to obtain a very accurate picture of the operations running inside the container. By supplementing traditional metrics (including CPU load, memory usage, network and block I/O, and number of running processes), which can also be captured at host level, a comprehensive overall level of information about the state of a container can be aggregated. In addition, it should be mentioned that the collected state information can be directly assigned to the respective containerized workload, since there are no possible concurrent activities caused by a guest operating system that could distort the overall picture. This type of monitoring allows a rule-based approach for the detection of common security events and attempted attacks of containerized workloads.

4 RELATED WORK

There are a number of methods for increasing the protection of containerized environments which improve the existing Linux security mechanisms that are nowadays supported by container runtimes. These include *SPEAKER* (Lei et al., 2017), which uses different seccomp profiles for the start-up phase of the container and its actual operational phase, or *LicShield* (Mattetti et al., 2015) which generates policies for AppArmor with the help of a behavioral learning phase.

Although approaches (Nikolai, 2014) exist in hypervisor-based environments that only use performance metrics to detect malicious behavior and would be transferable to containerized environments, system calls are still the preferred data source when it comes to detecting malicious behavior. This goes back to the seminal work of Forrest (Forrest et al., 1996), who proposed sequences of system calls as an appropriate differentiator between normal and anomalous behavior. However, the accuracy of this approach suffered from concurrency on the host causing distortions of results, such as parallel services. Containerized workloads reduce this risk, since only one application typically runs in one container which avoids concurrency noticeable.

Abed et al. (Abed et al., 2015) applies the approach of using *strace* to containers and found it suited to be used for anomaly detection without requiring any prior knowledge of the container. Unfortunately, the utilized bag of system calls approach is generally vulnerable to mimicry attacks (Kang et al., 2005).

Another paper dealing with the practicability of Falco is provided by Borhani (Alex Borhani, 2017), although it focuses on *Incidence Response* rather than the limitations of the approach.

The use of machine learning approaches to monitor workloads based on system calls is also investigated by a number of authors, including Maggi et al. (Maggi et al., 2010) using Markov models for anomaly detection based on system calls and their arguments. This approach was extended by (Koucham et al., 2015) by the incorporation of domain knowledge and system call specific context information. Kolosnjaji et al. (Kolosnjaji et al., 2016) described an approach with neural networks based on convolutional and recurrent network layers which allows them to improve the performance of detecting malware. The approach of Dymshits et al. (Dymshits et al., 2017), which focuses on distributed collection and processing of large volumes of data, investigates sequences of system call count vectors and uses a LSTM-based architecture.

5 OVERVIEW OF PROPOSED SYSTEM

Our approach is based on the Open Source tools *Sysdig* (Sysdig, 2019b) and *Falco* (Sysdig, 2019a). They both distinguish from other approaches such as using *strace* and *eBPF* (Fleming, 2017) by a native support

for containers (Docker, rkt, LXC). Their ability to activate filters to restrict the collection of data, primarily system calls, to single containers significantly helps to reduce the effort associated with container-specific monitoring and also offers the potential to reduce the amount of data that has to be collected and processed.

5.1 Sysdig

Sysdig consists of a Linux kernel module (*sysdig_probe*) and a daemon. Through the kernel module, all system calls coming from applications and containers are captured and sent to the daemon for further processing. The daemon can either be executed on the host system or within a privileged container. The functionality of Sysdig extends over several well-known analysis and monitoring tools, including *strace* and *tcpdump* and also includes transaction tracing. This allows capturing not only the state of the system (CPU load, running processes), but also actions as accessing files or establishing network connections. To limit the amount of data, it is possible to define filters for incoming events. These can be based on specific system calls, the source of an event, such as specific containers or processes, or attributes of the respective event.

5.2 Falco

Falco complements the system call capture functionality of Sysdig with the ability to detect abnormal activities. It is based on the same kernel module as Sysdig and implements a behavioral activity monitor whose policies can be defined based around the Sysdig filter options as *condition*. It can be used to identify, using an appropriate set of rules, whether a container deviates in its behavior from the desired state and shows signs of anomalous behavior. Such anomalous activities inside a container could represent starting unauthorized applications in a container, browsing unusual file system paths, or attempting to write to paths containing system files and binaries. Falco, however, only *detects* violations, so that mitigation of the detected anomalous behavior must take place subsequently. Another limitation of Falco is that it is only suitable for detecting point anomalies, such as the occurrence of a certain event, as starting of a certain application, or the use of a unauthorized system call. Currently there is no possibility to map contextual anomalies or collective anomalies through the available rules set. This limits the detection possibilities to some extent, as the following investigation shows.

5.3 Architecture

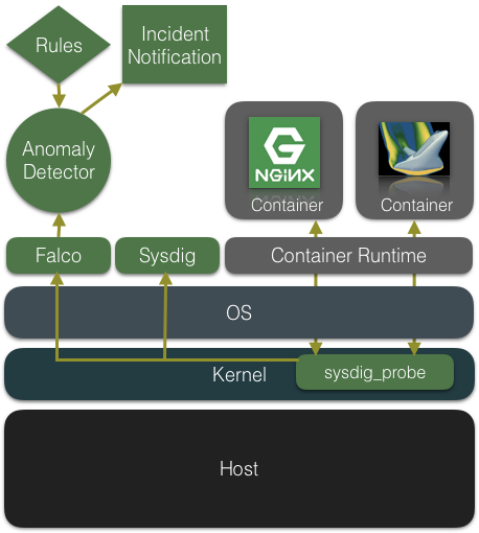

Figure 1: Proposed Rule-based Security Monitoring Architecture.

Figure 1 shows the corresponding architecture. As already described, both Sysdig and Falco use the (*sysdig_probe*) kernel module which utilizes the Linux kernel facility *tracepoints* to capture system calls. System calls are of high value when it comes to tracing the behavior of a (possibly containerized) application, since already every file access leaves a meaningful trace at the transition from the process-to-kernel boundary, also known as the system call interface. This way, besides the call names, also the arguments and the return values can be obtained and evaluated. While Sysdig is mostly used for displaying and processing captured events, Falco provides the possibility to decide on the basis of predefined rule sets (Figure 1, *Rules*) if there is anomalous behavior and to perform a corresponding notification (Figure 1, *Incident Notification*) using logging frameworks, messengers or plain e-mail. We are currently thinking about expanding this functionally and introducing mitigation measures based on the severity of the event.

6 EVALUATION

For our evaluation we have chosen a two-phase approach: in the first phase we manually examine whether and how certain behavior can be recognized with Sysdig and, if possible, transfer the condition that is needed to recognize the undesired activity into a Falco rule set. In the second phase the created Falco rule set was tested for the possibility of automatic recognition.

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:

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 we used Debian GNU/Linux 9.5 (stretch) as container image.

6.2 Studied Misuse and Attacks

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 [1] is *webgoat* and the system call *open* accesses a file outside a path containing the string */webapp*.

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

[1] 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.

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 Results

Table 1 shows the summary of the findings and also lists what can be used as a characteristic when creating the condition used for detection.

It should be noted that in almost all cases it is possible to create a rule set for Falco if a Sysdig condition allows detection of the event. As already implied upon presenting the test cases, it is straightforward to detect access to non-authorized files, the start of non-authorized applications and cases derived from those scenarios, either via the violation of a white list or explicitly via a black list.

The detection of a DoS attack can in principle be detected with Sysdig, but requires that it is combined with a different behavioral monitor than Falco, since it is not possible with Falco to detect, for example, high frequent occurrence of write access or network connections. There is no support for an event frequency as of writing.

Also the detection of a Buffer Overflow is not possible in the desired way, as it is typically also not detectable by static analysis. The execution of an exploit would generate multiple system calls. However, Falco can only detect anomalies from a single system call. In this scenario, the order and combination are also of importance and the approach to detect one specific exploit could not be generalized. As a workaround it would be possible to detect the startup of an exploit by process name. Yet this requires knowledge of the specific exploit to be able to detect its execution through a blacklist - and is not very promising either, as it could be bypassed by simply renaming the exploit.

7 DISCUSSION

According to our understanding, there are a number of restrictions on the tools examined which should not go unmentioned. This includes the restriction that there is no possibility to detect attacks that show up by an increased number of accesses, for example a DoS against a service or if a high volume of data is transferred automatically. Falco's current rule set does not provide the possibility to consider frequencies or load conditions, even if they could be determined in principle.

In addition, the rule creation procedure shows that precise knowledge of the services running in the containers must be available in order to enable effective security monitoring. This includes communication endpoints, required file paths and program names. In an environment where hundreds of similar containers are started fully automated, this can be irrelevant, but the necessary effort should not be neglected in smaller more individual environments. Especially when using containers for interactive work - which is not recommended. A further limitation concerns the white list approach, which does not protect against malicious code being disguised as a program on the white list when knowledge of the rules is involved.

On a positive note, the tools utilized can also be applied to other container environments, even though we have only examined the use with Docker here. The kernel module provides a generic interface, so that existing rule sets can be reused when changing container runtimes. Furthermore, from a security point of view, it may be worthwhile to monitor not only the individual containers running on a host, but at least container life cycle operations performed on the host itself, such as starting/stopping a container, to get a complete picture of the environment.

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

Table 1: Summary of the detectability of various misuse and attack scenarios using Sysdig and Falco.

Scenario	Sysdig	Falco	Characteristic
Unauthorized File Access	Yes	Yes	Violation of white list
Start of Unauthorized Applications	Yes	Yes	Violation of white list
Container Breakout	Yes	Yes	Black list - *nsenter* called - or violation of white list
Unexpected Network Connection	Yes	Yes	Violation of white list
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

Table 2: 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%

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 2.

8 CONCLUSIONS

The explored approach shows a general applicability of rule-based security monitoring for containerized workloads with low performance overhead. It allows a large number of undesired behavior to be detected with relatively low effort. Especially if containers are used as automatically orchestrated infrastructure components and are not used for interactive work, it is useful to create a rule for each container, which informs about the start of a program in the container other than the expected service. This can also be applied to file accesses.

In addition, logging mechanisms for container access should be established at host level (via monitoring the use of the commands *docker exec* or *nsenter* or the related system call *setns*) and the tools be integrated into appropriate incident response processes in order to timely initiate appropriate mitigation. In addition, for debugging and auditing purposes, it offers the possibility to log the complete run of a container to trace it afterwards. However, this should primarily be used for short-lived containers or additional filters should be applied, as the performance overhead analysis showed, that without filters almost 1MB trace file was generated per second.

In cases where it is not possible to create a simple set of rules, ideally by automatisms, further safeguarding mechanisms which monitor the behavior of a container at runtime and compare it with a reference model should be considered, as the investigated approach requires prior knowledge about the workload.

Future work is planed to investigate distributed security monitoring, especially of workloads that interact with each other in a distributed way. For this purpose Sysdig already offers interfaces for monitoring the container schedulers Kubernetes and Swarm, furthermore a commercial product. We also want to investigate how the generation of rule sets can be automated, as well as how additional misuses and attacks can be integrated.

REFERENCES

Abed, A. S., Clancy, T. C., and Levy, D. S. (2015). Applying bag of system calls for anomalous behavior detection of applications in linux containers. *2015 IEEE Globecom Workshops, GC Wkshps 2015 - Proceedings*.

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

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.

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

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, 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.

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

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

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, 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, 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.

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

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

Portworx (2018). 2018 Container Adoption Survey. Technical report.

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.

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). Docker Usage Report 2018 - An inside look at shifting container usage trends.

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

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].