Global Information Assurance Certification Paper

Anomaly Detection, Alerting, and Incident Response for Containers

GIAC GCIH Gold Certification

Author: Alex Borhani, r.alex.borhani@gmail.com
Advisor: Chris Walker
Accepted: February 19, 2017

Abstract

With the rapid adoption of containerized technologies to support the agile development and operations (DevOps) methodology, the necessity of formulating a comprehensive prevention, detection, and incident response (IR) security strategy in those environments is critical. Though various mechanisms exist to fulfill preventive strategies for containers, such as system hardening and continuously patching images, the need to implement similar levels of detection capabilities is also vital, particularly because many preventative security efforts are eventually neutralized or, worse yet, never implemented properly. By outlining the capabilities of several open source technologies, this paper will demonstrate the viability of detecting an anomaly, alerting on the presence of an anomaly, and facilitating IR to eliminate an anomaly within a containerized and orchestrated environment.

1. Introduction

1.1. DevOps Culture

Lean manufacturing practices, introduced and matured by Toyota Motor Corporation, were required for Toyota to compete with foreign automobile companies, which were significantly better resourced. Toyota was able to maximize effectiveness, efficiency, and economies of scale from personnel and resources through tightly managed processes and methodologies throughout the entire automobile production life cycle. What Toyota did with lean manufacturing practices to the automobile industry, is the equivalent of what DevOps is attempting to mature within the application development life cycle (Ries, 2011).

DevOps is the union between developers, the individuals who create the applications, and operations--the individuals who ensure the availability, performance, and functionality of the applications. The DevOps union introduces lean application delivery by maximizing productivity through automation, and by eliminating barriers and waste. Due to its adoption and success in some of the top technology companies, DevOps culture and practices will remain a common and popular software development lifecycle methodology.

1.2. Container Technology Overview

The term "container" became popular with the release of Solaris 10 operating system (OS). In Solaris 10, a container was an application with access to only authorized resources (Vaughan, 2009). However, this capability predated Solaris 10 because a similar functionality was natively available in the first UNIX OSs, where processes were "jailed" in isolation and prevented from accessing protected resources.

Though the term "container" was popularized in Solaris 10, the ability to create and manage containers remained a very difficult process. However, in 2013, the capability to effortlessly create and manage a container became viable using a new tool called Docker. Other modern container technologies have emerged after Docker's release; however, Docker remains the most popular container platform.

When operating a solution like Docker, one should think of containerization as if it's a shipyard, where shipping containers are methodically delivered, organized, loaded, shipped, and offloaded (Mouat, 2016). Docker explains that its platform allows "developers and IT admins to build, ship and run any application, anywhere" (Docker, 2017a). Hence, when using a modern container platform, not only are users able to facilitate application isolation, but they also gain access to an ecosystem capable of automating the delivery and management of that application.

Containerization and virtualization are two different technologies. Many IT professionals would associate the term "virtualization" as the ability to install an entire OS within a virtual machine (VM), with the VM hosted and managed by a hypervisor allocating the necessary physical hardware resources to meet the demands of the various VMs it hosts.

The benefit of utilizing containers versus VMs is that VMs require substantially more resources to be allocated to them as they must install an entire OS to support the installed applications. As containers require fewer resources, such as CPU cycles, RAM, and storage, compared to an OS, a developer can engineer, configure, and test several applications on a single OS instance, even though those applications perform completely different or unassociated services. The multi-container capability increases the productivity of a developer, who can now spend more time developing and testing applications versus having to configure and manage multiple VMs.

Furthermore, once a container completes testing, a developer can ship that container to the production environment and have the container seamlessly loaded into production, if containerization is available. Due to its flexibility and productivity, containers usage within the highly agile and lean DevOps culture is becoming the norm.

As VMs increase productivity out of computer hardware, containers provide a similar increase in productivity out of an OS. Though containerization can operate directly on non-virtualized hardware, many organizations will elect to layer VM and containerization technologies together to maximize productivity and availability of services. Regardless of VM and container strategy, the utilization of these technologies introduces new complexities and capabilities for organizations to capitalize.

Alex Borhani, r.alex.borhani@gmail.com

1.3. Orchestration Technology Overview

One key DevOps principle is to "automate anything" (Hering, 2015). The term commonly used to describe container automation is orchestration, a concept reminiscent of the assembly line processes used in automobile production, where some automated actions are taken based on environmental feedback. Some of the popular container orchestration solutions include Docker Swarm and kubernetes (k8s) (Ankerholz, 2016).

The term orchestration may have its roots in control theory; a theory about how dynamic systems can be influenced (Jacko, 2009). However, in DevOps culture, orchestration is primarily used to differentiate automation in containerized environments compared to automation facilitated through provisioning, to include deployment and configuration management.

Provisioning solutions like Puppet, Chef, and Ansible became very popular in facilitating automated configuration management, and remain popular tools in DevOps as they are still used to allocate and manage resources and support continuous delivery (Gibbs, 2015). Traditional provisioning solutions can still provide some orchestration functionality for containers but are not on par with more modern Docker Swarm or k8s implementations (Fiedler, 2015).

One way to distinguish provisioning and orchestration is that through orchestration there is intelligent functionality for when one container, i.e. service, gains a higher amount of network traffic, a second container of the same service can be automatically initiated to provide load balancing support. Afterward, when the traffic begins to decrease, the second container can be automatically terminated. Orchestration's capability to dynamically react to environmental changes in near real-time gains significant viability through the flexibility and lightweight nature of containers.

These four technologies: virtualization, provisioning, containerization, and orchestration, are independent of one another; however, by layering and integrating these solutions, an organization could very quickly build a highly available, scalable, and redundant platform, all while minimizing resource waste through automation.

1.4. Container and Orchestration Security Implications

With the advancement of the containerization and orchestration, gone are the days when a security team had to harden and monitor a single server with one operating system, which had one critical service running on it. An organization can now use a single server, hosting dozens of VMs, with each VM running hundreds of containerized business critical microservices, which are dynamically scaled to meet operational and business-related demands.

Traditional security solutions are difficult to execute in containerized environments. For example, though external network communications can generally be monitored using tools like network intrusion detection or prevention systems, internal communications through linked containers cannot be monitored as the activity does not traverse over the traditional network. However, container network monitoring can be facilitated via open source tools such as Moloch (Gomez, 2016), if deemed necessary.

Full packet analysis will serve as a comprehensive mechanism to monitor network activity. However, a similar strategy will be needed to monitor individual containers for potential compromise, synonymous to host intrusion detection systems (HIDS). Unfortunately, installing an HIDS on a container defeats the lightweight and single service notion of a container, compared to installing HIDS on a host environment. Without being able to install traditional security tools in containers, most container users rely on applying preventative security settings and configurations as their primary strategy.

2. Mitigating Unauthorized Access in Containers and Orchestraters

2.1. Preventative Security Settings

One of the many benefits of containerization and orchestration is the ability to automate the integration of security into the provisioning, releasing, and updating systems and applications. Hence, each unique OS and container image used in automated provisioning or orchestration should be preconfigured with preventative security settings.

Creating a hardened OS image should be the first step. Solutions like SELinux and AppArmor use access control mechanisms to harden a host Linux OS environment (Hayden, 2016). Furthermore, a system administrator can run and configure Grsecurity and PaX to apply additional security mechanisms during compile and run times on a Linux OS kernel (Docker, 2017a). However, these tools perform system call filtering/monitoring at the kernel level, which may result in difficulty in monitoring interactions with external sources such as k8s (Stemm, 2016b).

Creating hardened container images should be the second step. There are several preventative security settings available for containers, including placing the container in a read-only capacity, defining CPU and memory allowances, patching base images, user name spacing, Docker Notary, and applying encryption. However, not only is it critical to apply various security elements, but it's also important to establish proper accountability through auditing to ensure those settings are enforced properly (Robinson, 2016).

2.2. Detecting Security Anomalies

New vulnerabilities are identified in applications on a regular basis, so taking preventative measures to secure a system is not enough. Even with automated updating of applications and dependencies via orchestration, a patch to a publicly announced vulnerability may not release quickly enough. Furthermore, many security settings are not properly implemented, configured, or tested; hence, relying on preventative measures may create a false sense of security.

Exercise a proper defense in depth strategy to improve the security poster for containers and orchestrators. Though preventative strategies will mitigate the majority of known issues, detection capabilities must be in place to identify unknown issues. A detection security strategy will require mechanisms to monitor, alert, and investigate anomalous behavior through incident response (IR).

2.3. Anomaly Detection Framework

A proper anomaly detection framework for container and orchestrators will require three key requirements. The first requirement is having the ability to natively monitor containers for anomalous behavior. The second requirement is having the ability

to alert anomalous behavior. The final requirement is having the ability to respond to the alert by investigating and mitigating the anomalous behavior.

2.3.1. Performance Versus Security Monitoring

DevOps culture encourages the use of measurements to improve success incrementally. One of the ways to measure this success is through monitoring applications to determine business gains, customer experience, application/system performance, and software quality.

Container and orchestrators have native monitoring capabilities. For example, Docker command options “stats”, “top”, “logs”, “events”, and “exec” are very useful in tracking performance in containers (Borello, 2016). While on the orchestration side, k8s utilizes visualization solutions such as Kubelet and cAdvisor (Kubernetes, 2017a). Becoming familiar with the native container and orchestrator commands and tools are critical for IR purposes.

There are numerous tools available to support the DevOps application success measures. Open source solutions include InfluxDB and Grafana, Prometheus, and Heapster, while subscription solutions include New Relic, Dynatrace, and AppDynamics. Though these solutions may detect anomalous activity, they are mostly intended to determine application/container performance. The tools provide log collection, visualization of predetermined or customized metrics, and many provide alerting functionality (McKendrick, 2015).

There are several commercial solutions focused on container security as well, such as TwistLock, Conjur, and Banyanops (Fiedler, 2015). All three platforms support some combination of auditing, access management, or vulnerability detection. These solutions ensure that security best practices are in place as preventative measures. Of the three solutions mentioned, TwistLock also offers some detection capabilities, such as building behavior profiles for containers and if anomalies occur it can notify, log, block user access, or kill compromised containers (TwistLock, 2017). This type of functionality gives TwistLock both HIDS and host intrusion prevention system (HIPS) capabilities.

2.3.2. Sysdig and Falco Overview

An alternative open source solution to container monitoring and detailed troubleshooting is sysdig. Sysdig is capable of capturing system events, applying filters, and running useful scripts. It incorporates functionality found in other open source tools like, "strace, tcpdump, lsof, htop, and Lua" in a single platform. What distinguishes sysdig from other host troubleshooting tools is its native support for modern containers and orchestrators (Degioanni, 2016).

Sysdig is not inherently designed to detect anomalies in containers, but its collection, filtration, and scripting capabilities allow for an excellent framework to troubleshoot or conduct IR for containerized environments. However, before IR can initiate, detecting anomalous or unauthorized activity is necessary.

Detecting anomalous or unauthorized activity is left to another Sysdig company project called Falco, which serves as an open source solution that combines OSSEC and Snort capabilities for containers, and monitors for potential anomalies and unauthorized access (Sysdig, 2017).

2.3.3. Sysdig Falco vs. Kernel Level Detection Systems

Falco operates by leveraging sysdig's filtering capabilities to identify and alert on any type of pertinent behavioral activity. Furthermore, Falco natively supports container and orchestrator terminology and references used in those frameworks, such as container ids, k8s namespaces, image names, etc. Hence, when it alerts, it's able to properly cite the specific container or orchestrator process experiencing the anomaly.

Unlike kernel level detection solutions, Falco operates in user space which provides access to a more comprehensive set of information to power its detection rules. However, running in user space does have a drawback, in that it makes it more susceptible to tampering because the process can be killed or suspended (Stemm, 2016b). Hence, as previously iterated, the best security posture is a defense in depth posture, where the network, kernel, and user space monitoring is implemented.

Falco is able to analyze and correlate system calls in full context of how they perform by being built on top of sysdig's event and filtering libraries, and by operating in user space. For example, Falco sees the remote IP address accessing a particular process

name, as well as the process's parent or child processes, so it's able to provide those details in the alert (Stemm, 2016b).

2.3.4. Falco Detection Rules

Falco stores its rules in a YAML file, which is a "human friendly data serialization standard for all programming languages" (YAML, 2017). Each Falco rule consists of three elements: rules, macros, and lists.

The rule element has the following fields:

- rule: a unique name
- condition: a sysdig filtering expression applied to events to determine a match
- desc: a detailed description of what the rule detects
- output: the message output if a matching event occurs
- priority: the severity of the rule using the following categories: "emergency", "alert", "critical", "error", "warning", "notice", "informational", or "debug".
- enabled (optional): can be "true" or "false". If false, a rule will not load nor match against events. The default value is "true".

Macros are conditions that can be reused inside rules and other macros, minimizing the need to re-define common patterns. Macros can leverage sysdig filtering expressions to increase functionality.

A list is a series of defined items. Unlike rules or macros, lists cannot be parsed using sysdig filtering expressions. However, any other rule, macro, or another list can reference a pre-defined list. Lists allow Falco to eliminate having to maintain repetitive entries for items that are expressed regularly. One of the most practical uses for lists is when users want to define a series of authorized or unauthorized processes or system activities.

Figure 1 is an example of all the elements needed to operate a Falco rule to detect a shell spawning inside a container by a non-shell process:

```
- macro: spawned_process
condition: evt.type = execve and evt.dir=<
- macro: container
 condition: container.id != host
- macro: shell_procs
  condition: proc.name in (shell_binaries)

- rule: Run shell in container
desc: a shell was spawned by a non-shell program in a container. Container
entrypoints are excluded.
condition: >
        spawned_process and container
        and shell_procs
        and proc.pname exists
        and not proc.pname in (shell_binaries,docker_binaries)
output: "Shell spawned in a container other than entrypoint (user=%user.name
%container.info shell=%proc.name parent=%proc.pname
cmdline=%proc.cmdline)"
priority: WARNING

- list: shell_binaries
  items: [bash, csh, ksh, sh, tcsh, zsh, dash]
- list: docker_binaries
items: [docker, dockerd, exe, docker-compose]
```

Figure 1: A Sample Falco Rule (GitHub, 2017)

The above rule was color-coded to demonstrate how rules, macros, and lists can be interlaced to create powerful conditions to assist with detecting an anomaly within a Docker container. The yellow macro, "spawned_process", is filtering for any executed process. The green macro, "container", is filtering for containerized processes by excluding "host" initiated processes. The blue "shell_procs" macro is filtering only for process names contained in the red "shell_binaries" list. The red "shell_binaries" list defines the names of common Unix shell types. The "Run shell in container" rule will only trigger when the yellow "spawned_process" and green "container", and blue "shell_procs" macros are true. Furthermore, the spawned process must have a parent process, but the parent process is not one of the authorized processes listed in the red "shell_binaries" list or the auburn "docker_binaries" list. The benefit of this interlaced model is that once a macro or list is defined, all other rules, macros, and lists within the same YAML file can leverage the pre-define macro or list.

This section demonstrates how the first requirement of establishing an anomaly detection framework for containers can be met by using Falco to create dynamic anomaly detection rules which natively support and monitor container activity.

2.3.5. Falco Alerting Capabilities

The rule output from Figure 1 can be alerted using four different methods. Falco is capable of alerting through the standard output, a file, syslog, or a spawned program. The "falco.yaml" file is used to configure the alert channel(s). Standard output is the default configuration and displays the rule output on a Linux terminal screen. File output is designed to allow every alert to be appended to a file, in the same format as standard output. Syslog messages are formatted based on an organization's syslog daemon and are prioritized based on the rule's priority field setting. Program output may be the fastest way to integrate with an IR team, in that it can be configured to initiate any available program and write an alert to its standard input.

Figure 2 is a Falco program output alert example for sending an email:

```
program_output:
        enabled: true
        program: mail -s "Falco Notification" someone@example.com
```

Figure 2: A Sample Falco Program Output Alert (GitHub, 2016)

Additionally, all Falco alert mechanisms are capable of utilizing JSON output, a lightweight data-interchange format, which can be enabled through the falco.yaml configuration file or through the command line (JSON, 2017). If enabled, Falco will print a JSON object for each alert which contains the following elements: time, rule, priority, and output. JSON provides an easy and lightweight mechanism to transmit data between services, which allows Falco to integrate alerts with popular operations communication platforms like Slack and PagerDuty.

Figure 3 is a Falco program output alert example for posting to a Slack webhook:

```
json_output: true
…
program_output:
        enabled: true
        program: "jq '{text: .output}' | curl -d @- -X POST
        https://hooks.slack.com/services/{SLACK WEBHOOK}”
```

Figure 3: A Sample Falco JSON Alert to Slack (Stemm, 2016a)

This section demonstrates how the second requirement of establishing an anomaly detection framework for containers can be met by using Falco to create email or communication platform alerts.

2.3.6. Sysdig Capabilities (sysdig, csysdig, chisels and tracers)

As previously outlined, not only is sysdig a robust tool used to assist any Linux host IR activity, its native support for containers and orchestrators makes it a unique IR tool in containerized environments. To facilitate its comprehensive monitoring and troubleshooting capabilities, sysdig utilizes four distinct tools: sysdig, csysdig, chisels, and tracers. The sysdig tool offers native support for all Linux container technologies, including Docker, LXC, and rkt. It's capable of providing granular visibility into storage, network, memory, and processing subsystems. It also offers a filtering engine to help "dig" system information (Sysdig, 2017).

The csysdig tool leverages the sysdig collection system but displays the collected data in an intuitive and fully customizable curses-based user interface. Csysdig runs in its own container, or directly on the host, giving users visibility into every container operating on the host (Sysdig, 2017).

The chisel tools are written in Lua, a scripting language, and provide sysdig with the ability to analyze event data and perform certain actions, such as sending notifications when a particular action is completed or filtering through a tracer session (discussed later) to find a particular process. The Lua scripts are fully customizable; hence, chisels can be modified to meet any user's unique environment or operational need (Sysdig, 2017).

The tracer tools provide the ability to track and monitor any system activity as it processes through a system. Tracers begin at a point in time, which is called an "entry tracer", and end with a corresponding "exit tracer", which closes the tracking or monitoring. Tracers are fundamentally spans that capture anything that it's asked to capture. Tracers make it possible to capture network activity, similarly to what tools like tcpdump and Wireshark can accomplish, "so that the problem can be analyzed at a later time" (Sysdig, 2017).

2.3.7. Sysdig Incident Response for Containers and Orchestrators

Using the csysdig tool, an incident responder can graphically view all running processes on the host. When operating in a container environment, incident responders should run csysdig using the "-pc" option. If needing to connect to an external k8s's API, incident responders can use the "-k" option. Once csysdig is running, users should hit the F4 key and query for the name of the process that Falco alerted on. Once they hit enter, csysdig will filter the running process list to only the name query the incident responder conducted.

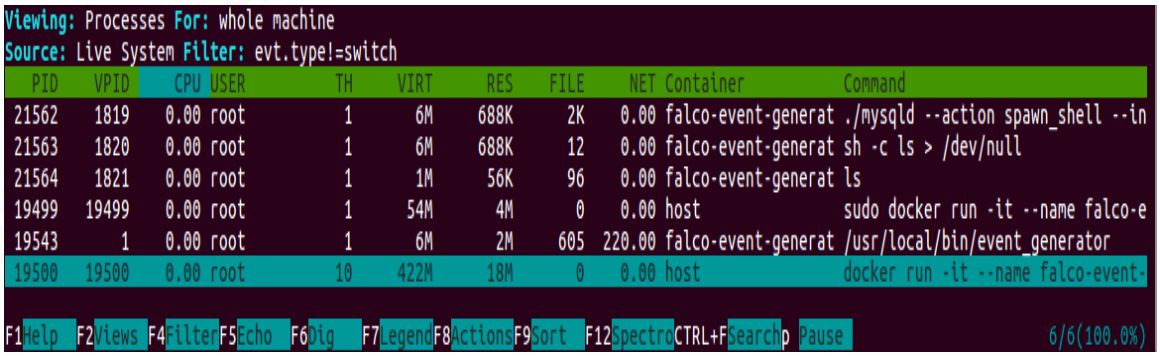

Figure 4: A Sample Csysdig Output With an Applied Text Filter

The keyboard's arrow keys highlight any process and the enter key can be used to drill down into that specific process. This technique can be repeated for sub-processes. This functionality is very useful for containers, in that using a single view, incident responders can quickly traverse from host level process to individual container level processes. Furthermore, depending on what users are reviewing, if they hit the F8 key, there will be several options available. For example, if an incident responder is reviewing

the container potentially with an unauthorized shell running, he/she could highlight the shell in question and initiate a ltrace on it or kill the process if deemed necessary.

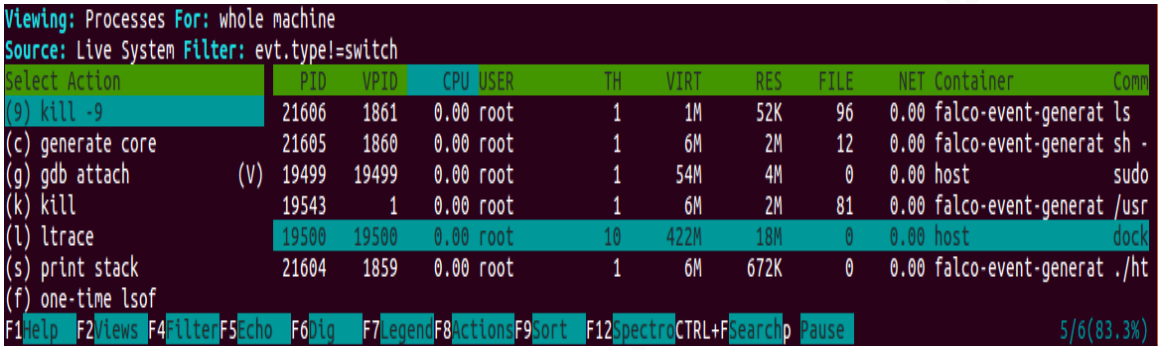

Figure 5: A Sample Csysdig F8 Options Output Menu.

The previous two sysdig sections demonstrate how the third requirement of establishing an anomaly detection framework for containers can be met by using sysdig to conduct incident response in containered or orchestrated environments.

2.3.8. Falco & Sysdig Logging Considerations

Ideally, a proper logging framework is in place to collect, parse, and store an organization's network, host, and application logs. If logging is not available, at a minimum, a syslog server should be implemented. A combination of Elastic Stack (formerly ELK Stack), Graylog, or Fluentd will serve as a superior open source logging framework alternative to syslog. There are several commercial security information and event management (SIEM) solutions as well, including Splunk, LogRhythm, and ArcSight. An open source logging framework or commercial SIEMs offer enhanced query engines, correlation, and data visualization capabilities, which a traditional syslog server cannot accomplish.

Most container and orchestrator solutions, including Docker and k8s, provide native logging capabilities. Docker provides a logging integration guide with several frameworks, including Fluentd (Docker, 2017b). K8s also has a logging integration guide for various platforms as well, including Elasticsearch and Kibana (Kubernetes, 2017b).

Falco natively supports syslog; hence, it can be integrated with most of the logging solutions mentioned above. Sysdig performs traces; hence, it can log any system level activity. However, sysdig requires some tuning to integrate with other logging

frameworks. One way to achieve this is to use the Linux "|" pipe function to send sysdig activity results to Logstash (Berman, 2016). To make this possible, incident responders should first install sysdig and Logstash, and then create a Logstash sysdig configuration file, which can be named "logstash-sysdig.conf". The configuration file needs to define stdin as the input, apply a grok filter to the data, and set the Elasticsearch instance as the output. Now, whenever a sysdig command is run, the results can be piped to Logstash using the "-f" option to load the settings from the Logstash sysdig configuration file. For example:

```
sysdig -t a "not(proc.name = sysdig)" | bin/logstash -f logstash-sysdig.conf
```

Figure 6: Sysdig Output to Logstash Command (Berman, 2016)

Piping is not limited to Logstash; hence, similar results are achievable by utilizing alternative log collectors such as Fluentd.

During an IR engagement, it is important to consider logging all IR tool activity. IR logging provides a way to keep a journal of investigative and forensic findings, and also allows other members of the IR team to track, review, analyze, and correlate IR tools' findings with other potential data.

3. Conclusion

Containers and orchestrators introduce a combination of new technologies and environments that may pose a challenge for many security teams. However, hosts and containers can be made secure using preventative security settings, such as system hardening, continuously patching images, and auditing. Furthermore, as preventative security settings could eventually falter, security teams can also consider establishing an anomaly detection framework to help identify anomalous activity occurring within containers. As demonstrated by this paper, security teams should utilize Falco to monitor for anomalies in containers, and alert the necessary IR personnel when anomaly detections are made. In addition, IR personnel should become familiarized with native container and orchestrator commands and solutions; however, as demonstrated by this paper, they should utilize sysdig as a primary IR tool due to its ability to natively

traverse, troubleshoot, and collect evidentiary data in containers and orchestrators. A combination of Falco and sysdig will serve as an ideal open source anomaly detection framework within containerized and orchestrated environments.

References

Ankerholz, A. (2016, April 12). 8 Container Orchestration Tools to Know. Retrieved August 22, 2016, from https://www.linux.com/news/8-open-source-container-orchestration-tools-know

Berman, D. (2016, October 19). Sysdig and ELK: A Match (Potentially) Made in Heaven. Retrieved February 7, 2017, from http://logz.io/blog/sysdig-elk-stack/

Borello, G. (2016). Monitoring Microservices: Docker, Mesos, Kubernetes Visibility at Scale. Velocity Conference 2016, Santa Clara, CA, USA June 20-22, 2016: O'Reilley Media, Inc. Sebastopol, CA.

Degioanni, L. (2016). The Dark Art of Container Monitoring. Open Source Convention 2016, Austin, Texas, USA May 16-19, 2016: O'Reilley Media, Inc. Sebastopol, CA.

Docker (2017a). Docker Documentation. Retrieved February 09, 2017, from https://docs.docker.com

Docker (2017b). Fluentd Logging Driver. Retrieved February 12, 2017, from https://docs.docker.com/engine/admin/logging/fluentd/

Fiedler, J. (2015, December 4). How to be Successful Running Docker in Production. Retrieve February 05, 2017, from https://www.youtube.com/watch?v=j6Ge4wP1yH0

Gibbs, N. (2015, April 13). What's Deployment Versus Provisioning Versus Orchestration? Retrieved February 05, 2017, from http://codefol.io/posts/deployment-versus-provisioning-versus-orchestration

GitHub (2017). Sysdig Falco. Retrieved February 10, 2017, from https://github.com/draios/falco

Hayden, M. (2015, July 26). Security Linux Containers. Retrieved February 03, 2017, from https://www.sans.org/reading-room/whitepapers/linux/securing-linux-containers-36142.

Hering, M. (2015, September 17). 8 DevOps Principles that Will Improve Your Speed to Market. Retrieved February 18, 2017, from https://www.accenture.com/us-en/blogs/blogs-devops-principles-improve-speed-market

Jacko, J. A. (2009). Human-Computer Interaction. 13th International Conference, HCI International 2009, San Diego, CA, USA July 19-24, 2009: Proceedings. Berlin: Springer.

Alex Borhani, r.alex.borhani@gmail.com

JSON (2017). JSON. Retrieved February 11, 2017, from http://json.org

Kubernetes (2017a). Resource Usage Monitoring. Retrieved February 09, 2017, from https://kubernetes.io/docs/user-guide/monitoring/

Kubernetes (2017b). Logging with Elasticsearch and Kibana. Retrieved February 12, 2017, from https://kubernetes.io/docs/user-guide/logging/elasticsearch/

McKendrick, R. (2015). Monitoring Docker. United Kingdom: Packt Publishing.

Mouat, A. (2016). Using Docker. Sebastopol, CA: O'Reilly.

Ries, E. (2011). The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. New York: Crown Business.

Robinson, A. (2016, November 18). A Checklist for Audit of Docker Containers. Retrieved February 02, 2017, from https://www.sans.org/reading-room/whitepapers/auditing/checklist-audit-docker-containers-37437

Stemm, M. (2016a, October 25). Find the Hacker. Retrieved February 05, 2017, from http://www.slideshare.net/Sysdig/find-the-hacker?qid=86aa74e1-2ef5-499a-abf5-57362f0d82d1&v=&b=&from_search=12

Stemm, M. (2016b, December 09). SELinux, Seccomp, Falco, and You: A Technical Discussion. Retrieved February 04, 2017, from https://sysdig.com/blog/selinux-seccomp-falco-technical-discussion/

Sysdig (2017). Sysdig Documentation Wiki. Retrieved February 08, 2017, from http://www.sysdig.org/wiki

TwistLock (2017). TwistLock Runtime Features. Retrieved February 08, 2017, from https://www.twistlock.com/runtime/

Vaughan, J. (2009, May 07). Solaris 10 Containers for OpenSolaris. Retrieved February 05, 2017, from https://blogs.oracle.com/lunchware/entry/solaris_10_containers_for_opensolaris

YAML (2017). The Official YAML Web Site. Retrieved February 10, 2017, from http://yaml.org