Forensicating Docker with ELK

Stefan Winkel
Forensicating Docker with ELK

GIAC GSEC Gold Certification

Author: Stefan Winkel, stefan@winkelsnet.com
Advisor: Dr. Johannes Ullrich
Accepted: 07/09/2017

Abstract

Docker has made an immense impact on how software is developed and deployed in today’s information technology environments. The quick and broad adoption of Docker as part of the DevOps movement has not come without cost. The introduction of vulnerabilities in the development cycle has increased many times. While efforts like Docker Notary and Security Testing as a Service are trying to catch up and mitigate some of these risks, Docker Container Escapes through Linux kernel exploits like the recent widespread Dirty COW privilege escalation exploit in late 2016, can be disastrous in a cloud and other production environments. Organizations find themselves more in need of forensicating Docker setups as part of incident investigations. Centralized event logging of Docker containers is becoming crucial in successful incident response. This paper explores how to use the Elastic stack (Elasticsearch, Logstash, and Kibana) as part of incident investigations of Docker images. It will describe the effectiveness of ELK as result of a forensic investigation of a Docker Container Escape through the use of Dirty COW.
1. Docker Introduction

Docker is a platform that combines applications and all their dependent components (e.g. libraries, tools) into an archive called a Docker Image. Users can run a Docker Image on many different platforms like PCs, data centers, VMs or clouds. As a Docker Image compartmentalizes the application(s) and all its dependencies, it provides various benefits over bare metal like portability and scalability. These features, combined with a reduced footprint that Docker Images have over Virtual Images, result in deployments of Docker Images in many different environments like data centers and cloud solutions.

Started in 2013, Docker is an open-source project and was released under the Apache 2.0 license, which efficiently allows for the creation, shipment, and running of containers within a single Linux instance. Docker was initiated as a project to build single-application Linux Containers (LXC) and introduced numerous improvements to LXC that make containers more flexible and portable to use compared to LXC, as well as some other older container technologies like FreeBSD Jails and Solaris Zones (Docker, 2016).

Docker has contributed significantly to changes in the development cycle for many organizations. In 2014, Amazon deployed 50 million changes: that is more than one change every second of every day (Brigham, & Liguori, n.d.). “Google spins up more than 2 billion containers per week, more than 3,300 containers per second.” (Beda, n.d.). The vast amount of changes that companies are making on a regular basis to their products, together with the widespread use of containers, have contributed to the explosive growth of Docker.

1.1. Kernel Usage in Docker

One major benefit of using containers over Virtual Machines (VMs) is that containers have less overhead associated with server density as they are typically $1/10^{th}$ to $1/00^{th}$ the size of a similar application packaged within in a VM. A technology called Linux Containers (LXC) achieves this reduced server density. In LXC, a Linux Kernel is

Stefan Winkel, stefan@winkelsnet.com
shared to manage the underlying Operation System (OS). If for example, a physical server would be running 4 VMs, this would require 4 OSes in addition to a hypervisor. But with containers like Docker, the server could share the same OS, binaries, and libraries as shown in ‘Figure 1: VMs and containers resource utilization comparison’ below:

LXC, based on a user-space lightweight virtualization mechanism that implements namespaces and Control Groups (cgroups), manages resource isolation. Namespaces handle isolation for a single process while Control Groups deal with the isolation for a collection of processes (Wang, 2016). Cgroups isolate and limit a given resource over a collection of routines to control performance or security.

Portability is one of the biggest advantages of Docker over LXC (Wang, 2016). Portability allows the container to run on different OS distributions and hardware configurations without any changes to the image itself. The portability of Docker is suitable to be used in a multitude of different architectures suitable in cloud environments. Though containers share the same Linux kernel, they are platform agnostic, which makes them portable to any environment. Other benefits of using

Stefan Winkel, stefan@winkelsnet.com
containers include encapsulation and scalability. Encapsulation means to package everything needed by the application (e.g. dependencies, environment variables) within the container. The containers are also scalable which means that they can be dynamically reduced or expanded in size. These powerful container concepts explain why there has been an immense growth in Docker usage in DevOps environments in the past few years.

1.2. Docker Attacks

With Docker gaining immense popularity there is a shift in the threat landscape as new attack vectors appear (Winkel 2017). Some of these vectors include poisoned images (e.g. running the attacker’s image), compromised secrets (e.g. API keys, user credentials) and Denial-of-service attacks leading to container starvation. Another vector is kernel exploits which might take down the entire host and all Docker containers on that host at once since all containers on that host share the same kernel among themselves including the Docker host.

Container breakouts are yet another attack vector when dealing with Docker containers. While security practitioners should not ignore the Docker daemon attack’s surface (e.g. configuration parameters on how to start the Docker daemon, host networking), it is somewhat easier to defend against with Docker security tools than Container breakouts. There are various security tools, like Docker Notary, Docker Bench, and Twistlock, for hardening the Docker image for many of these attacks. Unfortunately, none of them will detect zero-day Linux privilege escalation exploits. The impact of Container breakouts could be widespread as it potentially allows adversaries to take over the underlying host OS, or even worse it could compromise the entire environment.

1.2.1. Docker Container Escape

To escape out of a Docker container to the underlaying system, one or more of the following items needs be present:

- Kernel vulnerabilities. An exploitable issue in the kernel can be used to break out of the container since host and containers share the kernel.
• Incorrect configuration. If a container is running with --privileged the attacker is likely to be able to get access to the underlying host.

• Mounted filesystems. If a container mounts a host filesystem, the attacker can plausible mutate items in that filesystem which could allow him to escalate privileges to the host. There are several dangerous locations in /proc and /sys that avow such trivial container escapes. Such simple exploits include changing the position of a utility (such as modprobe) that the host will call in certain scenarios (such as a kernel module load request). By changing this, an adversary could cause the host to run an arbitrary command outside the container.

• Mounted Docker socket. A common (and incorrect) practice in Docker containers is to mount the Docker socket inside a container, to allow the container to identify the state of the Docker daemon. Such usage allows a simplistic breakout to the host.

In 2014, a Docker container breakout attack by the name of Shocker was published by Sebastian Krahmer (2014). This exploit made use of a feature, included in Docker by default, called CAP_DAC_READ_SEARCH. This file capability feature allows a process to bypass both a file and directory ‘read’ and ‘execute’ permissions. While intended for searching, reading files or both, it also granted the process permission to invoke the system call ‘open_by_handle_at’. This call allowed a process to view the contents of any file using the inode value, bypassing namespace restrictions and enabled adversaries to break out of the Docker containers.

At Blackhat Europe 2015, Anthony Bettini presented a critical Docker privilege escalation bug (Bettini,2015). Bettini, the founder of the company Flawcheck, stated that among the numerous CVEs reported against Docker, CVE-2014-9357, a privilege escalation via exploiting a bug in the parsing of XZ files was probably the highest ROI attack vector for privilege escalation against Docker until that point (Bettini, 2015). Bettini described that Shellshock is also particularly relevant for Docker containers as most Docker containers contain bash. Since old copies of bash are typical, it is entirely possible to exploit the container if the running process in the Docker container uses bash.

Stefan Winkel, stefan@winkelsnet.com
While there are various root causes that can lead to a Docker container escape, configuration issues are relatively easy to detect and prevent by use of Docker security tools like Docker Bench which implements checking against many of the hardening recommendations as recommended by Docker CIS Benchmarks (Center for Internet Security, 2017). Container escapes through Kernel exploits are much harder to detect and prevent, even with a container vulnerability analysis service like Clair, especially since those types of tools won’t detect zero-day exploits ("GitHub - coreos/clair: Vulnerability Static Analysis for Containers," 2015). Hence proper logging and monitoring the behavior of Docker Containers for this particular type of Container Escapes is crucial.

1.3. Mitigate Container Escape: Segregation

To alleviate the increased risk with a shared Linux kernel between a container(s) and host in a Docker deployment, the principle of defense in layers should be applied by segregation. In a multi-tenant configuration, segregation can be implemented by placing users on different VMs and or hosts. Adrian Mouat, chief scientist at Container Security and author of the book Docker Security, suggests that “each user is placed on a separate Docker host” (Mouat, 2016). While it is less efficient than sharing hosts between users as it results in more VMs or bare metal, it is important from a security perspective. According to Mouat’s vision, a majority of Docker container deployments will involve VMs in the foreseeable future. While far from ideal, it balances the efficiency of containers with the security advantages of VMs. Figure 2 shows how container segregation by hosts for a multi-tenant setup could look:

Stefan Winkel, stefan@winkelsnet.com
Container segregation, through the use of bare metal hosts and or Virtual Machines for different users or business logic, reduces the impact of a potential compromise of the Docker host.

1.4. **Mitigate Container escape: Monitoring**

Since containers on one host are sharing the same kernel as the host, an exploitable issue in the kernel could be used by adversaries to break out of the container. The following conditions must be present to escape:

1. A vulnerable kernel
2. A matching exploit
3. The option to transfer the exploit onto the vulnerable system
4. The potential to execute the exploit on the target

Stefan Winkel, stefan@winkelsnet.com
Administrators can reduce the chances on container escapes by focussing on restricting or removing programs that enable file transfers, such as FTP, TFTP, SCP, Wget, and Curl (Long, 2016). If needed, use of these tools should be limited as well as access controlled to users, locations, applications, IPs, and domain, or both. Supplementary activities of users which the permissions should be logged and monitored.

Though various factors need to be present to escape out of a Docker container, it is relatively easy and hard to prevent and it can have a disastrous impact when not early detected and mitigated.

2. Docker Escape with a recent Linux Kernel Exploit

In this chapter, a more recent and widespread Linux kernel exploit, Dirty COW, is described as well as a variant on Dirty COW, called vdSO. This one has no fancy name and will be referred to as the vDSO exploit in this article. vDSO, a Linux kernel system, stands for virtual dynamically shared objects and is used to reduce system kernel calls. Since it is a recent and large-scale exploit, the vDSO exploit will be used to explore some of the monitoring requirements related to Docker container escapes.

2.1. Dirty COW

The “Dirty COW” kernel exploit received his name from the Linux concept, Copy on Write and refers to a concept where the kernel only writes memory pages back to disk if they do change (Long, 2016). Dirty COW was discovered in the wild in October 2016 through an HTTP packet capture by security researcher Phil Oester. The exploit was headline news because of its severe and widespread impact on millions of Linux devices as virtually all version of the Linux operating system contained this nine-year-old bug (Goodin/Ars Technica, 2016).

Shortly after Oester’s discovery, open source software vendor Red Hat put out a warning and described Dirty COW, CVE-2016-5195 as “a race condition in the way the Linux kernel's memory subsystem handled the copy-on-write (COW) breakage of private

Stefan Winkel, stefan@winkelsnet.com
read-only memory mappings “(Red Hat, Inc, 2016). This race condition occurs as the kernel uses an internal memory concept, called Copy On Write (COW), where operations that make a copy of a section of memory do not get their copy unless they make a change to that memory. The issue occurred when the memory is flagged to be writeable before it is copied. As this process occurs quickly over thousands of iterations, a race condition exists where the kernel overwrites the original file by mistake. Because of this vulnerability, adversaries could overwrite nearly any file, like ‘/etc/shadow’ or any other section of memory of their choosing without restriction. To summarize: an unprivileged local user could escalate his/her privileges by using this flaw to write malicious code into privileged files which can then be executed under the context of the root account to escalate privileges.

Besides that, Dirty Cow is a very recent Linux exploit, it is also a much more dangerous exploit than the other container escapes described earlier. Dan Rosenberg, a researcher at Azimuth Security, declared Dirty COW as “the most serious Linux local privilege escalation exploit ever” (Goodin, 2016). Rosenberg calls this vulnerability the proverbial ‘big one’ as the vulnerability exists in virtually every distribution of Linux. According to Security Focus, more than 900 Linux versions are vulnerable to Dirty COW (Security Focus, 2016). Linux Torvalds, author, and maintainer of the Linux kernel stated that it was an old bug and existed as early as 2005 (Torvalds / Linux Foundation, 2016).

The fact that it was an old bug indicates that the Dirty COW exploit could have been in use by adversaries for years before Oester's discovery. Because the exploit is recent, widespread, and potentially has a devastating impact, it is a great example to use when focusing on how to detect kernel escapes when exploring Docker monitoring.

### 2.2. vDSO exploit, a Dirty COW variant

As usual with large scale exploits, many proof-of-concepts of the Dirty COW vulnerability were quickly released ("PoCs · dirtycow/dirtycow.github.io Wiki · GitHub," n.d.). Shortly after the discovery of Dirty COW, an exploit variant on Dirty COW, that

Stefan Winkel, stefan@winkelsnet.com
uses ptrace and vDSO instead of /proc/self/mem and a setuid binary, was released by someone that goes by the handle Scumjr (2017). Stefano Stabellini, Linux kernel lead at the company Aporeto, describes the vDSO exploit in a blog article to be working from within any Linux container (Stabellini, 2016). This exploit will be used further in this paper to explore Docker logging and monitoring, and its functionality is described in detail in the following section.

2.2.1. vDSO exploit details

Libraries like glibc, and many userspace programs frequently execute kernel system calls like gettimeofday. Linux uses a mechanism called vDSO to minimize the overhead of kernel system calls. vDSO reduces the execution time of a small set of system kernel calls, typically timekeeping calls like gettimeofday. Programs call a routine at the right memory location, and the current time is returned without making any system calls.

Ptrace, a debugging and performance measurement tool, is a kernel hook into the task dispatch logic, which can be used to “trace” another program. The program allows the caller to trace the execution of a target process and copy data to a given memory address of the target process.

The vDSO exploit works as following, as described by Stabellini: “it opens a server socket, then spawns two threads. One thread keeps calling madvise (MADV_DONTNEED) giving the kernel advise how to handle paging and another thread concurrently attempts to modify the vDSO area using ptrace” (Stabellini, 2016). The exploit launches the payload that tries to open a socket and execute arbitrary commands. After that, it listens for incoming connections.

The exploit succeeds to modify the vDSO area because of the DirtyCow Linux kernel vulnerability. Since the kernel and program in user space share vDSO, eventually a root process on the host will try to obtain the system time. Once this occurs, the modified vDSO triggers the payload which makes a network connection is made to the listening socket in the Docker container, after which the adversary can run his or her commands at will. Figure 3 ‘vDSO exploit’ shows a graphical representation of the working of this exploit:

Stefan Winkel, stefan@winkelsnet.com
Essentially the vDSO exploit is relying on the same madvise system call with MADV_DONTNEED flag set, telling the kernel the user does not intend to use the memory. Instead of using a root-owned setuid binary it is relying on the vDSO-based root. The security community has collected over 20 similar Dirty COW proof of concepts on a Github page ("PoCs · dirtycow/dirtycow.github.io Wiki · GitHub," n.d.). The large amount of PoCs for this exploit is an indication for the severity and the criticality of this vulnerability.

According to Long, similar critical vulnerabilities like Dirty Cow are still present in the Linux kernel. Similarly, current privilege escalation techniques will remain viable for quite some time (Long II, 2016). As explained above, Dirty COW and the vDSO exploit are the most recent examples of ‘big one’ that potentially can cripple an entire organization. As similar Linux kernel vulnerabilities will continue to surface in the near foreseeable future, it becomes important to pro-actively detect container escapes as the Docker continues to advance the software containerization movement. The likelihood of another Linux kernel exploit like Dirty Cow is being discovered soon is very high. The next section explores how such kernel exploits potentially can be detected with a Docker Logging framework.

Stefan Winkel, stefan@winkelsnet.com
3. Docker Monitoring, Logging and Alerting

As Docker continues its explosive growth, support for Docker logging has evolved similarly rapidly over the past two years. Various companies have started to build commercial and open source solutions around monitoring and logging Docker containers as the need for comprehensive and centralized logging increases. Since Docker is a highly active software project, the logging framework explored here will focus on open source components.

3.1. Metrics versus Logs

For administrators and security personnel of an organization to understand what to log and where to log Docker events, it is important to realize the differences between metrics and logs. Brian Brazil, a developer at Prometheus, describes the differences as following, “metrics, also called time series, handle events aggregated across time. Logs, sometimes called event logs, are all about the context of individual calling an endpoint. Logs make the opposite to metrics” (Brazil, 2016). So, logging is more expensive as there is no aggregation over time. According to Brazil, this means that logs are limited to track around 50-100 pieces of information per event before cost and bandwidth are becoming a bottleneck.

Both metrics and logs have a place in a Docker logging framework but it will require careful considerations on what and where to log Docker events and where to keep the context for potential forensics analysis.

3.2. Contextual logging

Besides the fact that logs are more expensive than metrics, there is more to log when using Docker containers as they present various problems in terms of data proliferation. Compared to traditional stacks, there are more containers per host to monitor, and the number of metrics per host has increased (Hecht, 2017). As CoScale CEO, Stijn Polfliet, describes it, there would traditionally be 150 metrics to track per host: 100 about the operating system and 50 about an application. With containers, the administrator is adding 50 metrics per container and 50 metrics per orchestrator on the host. Considering

Stefan Winkel, stefan@winkelsnet.com
a scenario where there a cluster is running 100 containers on top of two underlying hosts, there would be over 10,000 metrics to track as shown in Figure 4 below (Hecht, 2017).

<table>
<thead>
<tr>
<th>Component</th>
<th># of Metrics for a Traditional Stack</th>
<th>for 10 Container Cluster with 1 Underlying Host</th>
<th>for 100 Container Cluster with 2 Underlying Hosts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>100</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>Orchestrator</td>
<td>n/a</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Container</td>
<td>n/a</td>
<td>500 (50 per container)</td>
<td>5,000 (50 per container)</td>
</tr>
<tr>
<td>Application</td>
<td>50</td>
<td>500 (50 per container)</td>
<td>5,000 (50 per container)</td>
</tr>
<tr>
<td><strong>Total # of Metrics</strong></td>
<td><strong>150</strong></td>
<td><strong>1,150</strong></td>
<td><strong>10,250</strong></td>
</tr>
</tbody>
</table>

*Figure 4: Per Host Metric Explosion (Source: TheNewStack_Book5_Monitoring_and_Management_with_Docker_and_Containers.pdf)*

To summarize, with Docker Containers the need for additional metrics and logs increases manifold.

Honeycomb co-founder and engineer Charity Majors explains in a recent blog post that traditional metrics lack context and are insufficient for tracking down complex intersectional root causes without context. She explains, “Metrics are usually bucketed by rollups over intervals, which sacrifices precious detail about individual events in exchange for cheap storage” (Majors, 2016). It is like Netflow logs without full packet captures. It is cheaper to log but in many instances, does not provide enough detail for detailed forensic analysis. In many cases, companies have lots of metrics but do not utilize them as they turn out not to be useful without context.

Users may have tens of thousands of containers, with some lasting only a few seconds. Traditional monitoring solutions have sampling and reaction time measured in minutes and do not scale for today’s environments. While metrics solve many operational problems, they do lack context. It’s important to have the contextual information about these containers to handle them in an environment where orchestrators are continuously spinning overwhelmed by metrics that aren’t useful (Degionanni, 2017).

Stefan Winkel, stefan@winkelsnet.com
Alex Williams, founder and chief editor of The New Stack, describes in his book, ‘Monitoring and Management with Docker and Containers’ that new logging tools are needed which are capable predicting what is going to fail (Williams, Ball, Dinh, & Hecht, 2016). Williams states there are three focal points for renewed monitoring: the health of container clusters; microservices; and applications.

Within the microservices as a focus point, there are four key signals to measure health and performance, according to the Google’s book on Site Reliability Engineering: latency, traffic, error, and saturation. Latency is the time it takes to service requests. Latency is useful to track slow API calls within a Docker container. Traffic and errors refer to the interaction of services and frequency of errors. These two signals are often tracked together. Saturation identifies the most constrained resources and utilization of the service (Beyer, Jones, Petoff, & Murphy, 2016).

To summarize, to detect potential malicious behavior Docker containers are requiring a need for renewed monitoring, as the number of metrics to track has exploded, and hence the need to have contextual information becomes more crucial.

3.3. Docker Logging frameworks

There are many options for monitoring Docker containers, ranging from free options, docker stats, CAdvisor, Prometheus or Sensu, to paid services such as Scout, Sysdig Cloud, and DataDog. Rancher Labs considers the following items when choosing a solution (Ismail / Rancher Labs, 2016):

- Easy of deployment
- Level of detail of information presented
- Level of aggregation of information from entire deployment
- Ability to raise alerts from the data
- Ability to monitor Non-Docker resources
- Cost

The Docker logging framework described in this article is a combination of various open source components described below.

Stefan Winkel, stefan@winkelsnet.com
3.4. **Docker Monitoring: Prometheus**

Prometheus is one of most foremost tools when it comes to metrics collection (Ismail / Rancher Labs, 2016). It is a pull based server which means the clients will need to provide a web interface from which it can scrape data. Various exporters are available for Prometheus which will capture metrics and then expose them over HTTP for Prometheus to scrape. As the objective is to monitor Docker containers, the Container Advisor (cAdvisor) exporter is used to capture Docker information. Grafana is an open-source front-end for Prometheus. Administrators can customize different dashboards by providing JSON files.

3.5. **Docker Logging: Elastic Search, Logstash and Kibana**

In considering a log management solution for Docker containers, organizations should take the following requirements into account:

- Provides lightweight log collection that does not interfere with the actual running of our hosts and applications (It should ship data quickly and avoid queuing crucial information.
- Needs to be a scalable and high-performing platform capable of processing large volumes of logs and storing them for an appropriate retention period to provide diagnostic value.
- Needs to be to parse and manipulate logs. The logging format requires the ability to normalize aspects like time and date, tags, hosts, and other information.
- Needs to integrate with the organization’s specific requirements, especially to pass events and metrics.

The Docker logging framework used here is based on the Elastic stack, also known as the ELK stack and is made up of three components:

- Elasticsearch — A document search store.
- Logstash — A log-routing and management engine.

Stefan Winkel, stefan@winkelsnet.com
• Kibana — A web-based dashboard and visualization tool

ELK is fast, relatively easy to set up, and is modular and flexible. It allows collection from many sources and can transform and normalize logs. Elasticsearch has proven to be a rich, searchable storage system, while Kibana is an excellent visualization interface. ELK has demonstrated to be useful in forensic investigations. Phil Hagen, an instructor at SANS, maintains a VM distribution SOF-ELK for the forensics community and uses it as well in the network forensics courses he teaches at SANS (Hagen, 2014). Instead of using the SOF-ELK VM, the ELK stack used here is run inside a Docker container itself which is ran inside an Ubuntu VM. There is ongoing discussion whether deploying ELK on Docker is an acceptable solution for production environments (resource consumption and networking are the main concerns), but it is sufficient to test the use cases in this article. One of the reasons for using a Docker image is to have additional Docker logs and events from the monitoring and logging framework itself. Another reason is that it allows for a simple extension of the monitoring component as explained below. To summarize, Prometheus will excel at time series manipulation, while ELK (and Lucene, the search engine under the hood) does an excellent job at free form data search and analysis.

3.5.1. Log framework with Machine Learning

Many of the new logging tools today are using artificial intelligence (A.I. or machine learning) techniques to identify patterns and detect anomalies (Williams, Ball, Dinh, & Hecht, 2016). Big Panda, CoScale, Dynatrace, Elastic Prelert, IBM Bluemix, Netsil and SignalFx are just a few of the companies that use artificial intelligence to identify patterns and detect anomalies. Peter Arijs of CoScale says anomaly detection means users don’t have to watch the dashboards as much (Arijs / CoScale, 2016). The system is supposed to provide early warnings by identifying patterns of behavior among how different services, applications and infrastructure behave.

3.5.2. A.I. with the Elastic stack

Stefan Winkel, stefan@winkelsnet.com
An example that provides machine learning capabilities is the Elastic Prelert. This commercial offering based on the ELK stack allows customers to deploy use cases that detect behaviors associated with different threat indicators like data exfiltration, compromised endpoints, unusual network activity, attacking IP addresses, etc. Each detected anomaly is assigned a normalized anomaly score and is annotated with values of other fields in the data that have a statistical influence on the anomaly. Elementary attack behaviors that share common statistical influencers are linked together into anomaly groups called ‘insights’ (Prelert, n.d.). As this is a commercial product, DMLA will not deploy Elastic Prelert.

Another, open source, example, is Xpack. Elastic stack has an Extension Pack that delivers machine learning capabilities. Currently, the AI functionality of Xpack is in Beta release. The X-Pack machine learning features automate the analysis of time-series data by creating accurate baselines of normal behaviors in the data and identifying anomalous patterns in that data (ElasticSearch, BV, 2017). The AI algorithms are proprietary and claim to detect the following circumstances, anomalies related to deviations in values, counts or frequencies, statistical rarity, and unusual behaviors for members of a population. Automated periodicity detection and quick adaptation to changing data ensure that users don’t need to specify algorithms, models, or other data science-related configurations to get the benefits of machine learning (Elastic, n.d.).

Critical to the success of the business when dealing with (micro) services at scale is the capability to monitor and have visibility into the organization’s infrastructure. As companies are iterating more quickly on products and services, key differentiation is visibility in their environments, so that development and operational teams can confidently and aggressively push the boundaries of innovation, knowing that when something breaks or get compromised, it will be quickly mitigated, before it damages the business and impacts customers.

The need for visibility means that monitoring and alerting will need to become smarter to provide continued value to the organization. As James Turnbull concludes in

Stefan Winkel, stefan@winkelsnet.com
his book, “alerting will need to be annotated with context and recommendations for escalations. Systems can reduce the number of unimportant alerts, which mitigates alert fatigue and increases the likelihood that the important alerts will be addressed” (Turnbull, 2016).

With the explosion of log data as a result of Docker container usage in modern production environments, monitoring and alerting will need to become smarter to avoid alert fatigue in order to be efficient in detecting potential malicious activities like Docker Container escapes.

4. Setting up the Test Environment: DMLA

As described above, Prometheus, with Grafana as a front-end, currently excels at time series manipulation, while ELK is preferred for free form data search and analysis. Hence, to get the best of both worlds DMLA is deploying both Prometheus as well as ELK. DMLA will use Wilhelm’s Uschtrin’s Docker in Box framework for the Prometheus functionality (Uschtrin, 2016). The following open source components as described by Uschtrin will be installed as part of DMLA:

- Monitoring: cAdvisor and node_exporter for collection, Prometheus for storage, Grafana for visualization.
- Logging: Filebeat for collection and log-collection and forwarding, Logstash for aggregation and processing, Elasticsearch as datastore/backend and Kibana as the frontend.
- Alerting: Elastalert as a drop-in for Elastic.io's Watcher for alerts triggered by certain container or host log events and Prometheus' Alertmanager for alerts regarding metrics.

Logging for both the Docker host as well as the Container logging will be handled by the ELK stack while monitoring of the host, as well as the containers, is done by Prometheus.

Stefan Winkel, stefan@winkelsnet.com

© 2017 The SANS Institute
• Dockerhost-Logging: Running a Filebeat container on a host suffices to forward all host logs to a centralized Logstash instance, which processes and forwards the logs to the Elasticsearch

• Container-Logging: Defining gelf as a logging driver for a container together with the logstash IP is enough for all logs that are usually going to stdout and stderr to be forwarded to Logstash

• Dockerhost-Monitoring: Just like running a Filebeat container for logging, running a node_exporter container on the host is enough to expose all host metrics for scraping by Prometheus.

• Container-Monitoring: cAdvisor aggregates metrics from all Docker containers running on the host and exposes them for Prometheus in the same way node_exporter does for the host.

Figure 5 represents a graphical overview of the various components of the DMLA framework that is being built:

![Diagram](image.png)
4.1. DMLA Setup and Configuration

To simplify installation DMLA will be deployed as a Docker environment. To minimize the overhead of downloading and installing DMLA users can download the latest code from Github.com at stefanwinkel/dmla. See also https://github.com/stefanwinkel/dmla and in Appendix Section A: ‘dmla_config script’ and Appendix Section B: ‘Building DMLA’ for more details.

To make sure DMLA is properly installed, use the following steps:

1) Generate a few NGINX logs for the ELK stack by browsing to the Nginx instance at

   1. http://localhost
   2. http://localhost/test123 (will show 404 not found error)
   3. http://localhost/failed456 (will show 404 not found error)

2) Login to the Kibana instance at http://localhost:5601 (username elastic, password changeme)

3) In Kibana, under the Discover tab on the left, search for the filebeat-*. A few Nginx logs should be displayed as shown in Figure 6.
4.2. Setting up the vulnerable Docker image

To minimize the impact of the Docker container escape and to minimize any other potential malicious behavior, the vulnerable Docker image should be run in a separate VM. But for networking simplicity reasons and since the DMLA Docker is run inside a VM the vDSO Docker image is executed within the same DMLA VM. Details on how to install this image are in Appendix Section C: ‘Installing the vulnerable Docker image’.

The vulnerable Docker image has been extended with various DMLA components, like FileBeat, similar to the Nginx image, as setup as part of the DMLA installation. This will help to identify the privilege escalation and Docker breakout at runtime by the DMLA framework. Similar to DMLA, the vulnerable image can be installed by running the ‘vagrant up’ command as shown below. The command is launched from the location where the Vagrantfile for the vulnerable image is located, in the ‘dmla/test_3’ subdirectory. The command will install an Ubuntu Virtual image (v14.04) after which a vulnerable Docker Image gets created. The ‘servers.yml’ script does the actual heavy lifting of constructing the VM, creating the Docker Image as well as compiling the exploit and transferring it to the Docker container.

Stefan Winkel, stefan@winkelsnet.com
In ‘dmla/test_4’, the Docker image is instrumented with various DMLA components to log and monitor the events before, during and after exploitation. See the ‘docker-compose.yml’ in this directory for details. Figure 7, Appendix Section D, shows the various commands that are run as part of the instrumentation. For simplicity reasons, the exploit is already transferred to the vulnerable Docker image. In a real-world scenario, the adversary might have compiled the exploit on a remote system and then transferred it to the vulnerable Docker image, when there is no compiler available on the vulnerable system. The attacker then could have downloaded the exploit to the Docker image with a scripting language like Perl to copy the 0xdeadbeef exploit to the Docker image. Note that while Scp, Ftp, Php and Python are not part of the Docker image and hence not available to the attacker, the image does contain Perl, which the adversary could use to transferring the exploit to the victim’s system. In this case, the exploit has already been made available for the attacker as ‘/dirtycow-vdso/0xdeadbeef’.

4.3. Breaking out of the Docker Container

To run the exploit in the vulnerable Docker Container, the following command is executed. See Figure 9 in Appendix Section D for contents of this file.

```
vagrant@dmlav1:~/dmla-v1/test_4$ launch_vdso_exploit.sh
```

To login in to the vulnerable Docker container execute the following:

```
vagrant@dmlav1:~/dmla-v1/test_4$ sudo docker-compose run
   dirtycow /bin/bash
```

When the DMLA framework is up and running, the exploit can be run from within the Docker container by executing the following command:

```
hacker@5a2e43e94153:/dirtycow-vdso$ cd /dirtycow-vdso && make && ./0xdeadbeef http://127.18.0.2:1234
```

Stefan Winkel, stefan@winkelsnet.com
4.4. Searching for Key Indicators of Compromise

As described earlier, to escape out of a Docker container the following four conditions must be present:

1. A vulnerable kernel
2. A matching exploit
3. The option to transfer the exploit onto the vulnerable system
4. The potential to execute the exploit on the target

Besides a vulnerable kernel, the exploit needs to be compiled for the correct platform. Furthermore, the adversary should have ‘write’ permission on the target system to write and execute the exploit from. If there is no compiler available on the target Docker image, he or she should be able to compile the exploit on a similar system and transfer it to the target vulnerable Docker image. Both the logging component as well as the monitoring aspect of DMLA can detect the vDSO exploit as shown in this paragraph.

4.4.1. vDSO system logs in the Container Group in Kibana

The administrator can see the various actions the user performed by searching through the logs produced by the vulnerable Dirty COW Docker image, referred to as the VulnImage below. Kibana, part of the DMLA framework shows the system logs of the VulnImage in Figure 7 below. The Docker container with name ‘test4_dirtycow_run_3’ is highlighted.
In the test case, the VulnImage is part of its own Docker container group, called ‘testing’, separately from Docker Images that are part of the logging and building container groups.

### 4.4.2. Unexpected and or malicious operations

By searching through the VulnImage system logs in Kibana the administrator can review the different commands that were executed by the attacker. The administrator can see that the attacker tried to execute the exploit but execution initially failed with a

---

**Figure 7: Container Group and Name in Kibana Index**

Stefan Winkel, stefan@winkelsnet.com
segmentation fault (e.g. core dump) as shown in Figure 8.

Also, the adversary tried to run a command as superuser, by running the ‘su’ command as seen in Appendix E, Figure 13. Figure 14 shows the system logs in DMLA that identify the attacker trying to download the vDSO exploit via Perl. In production scenarios, the administrator could log such operations and/or send alerts if Docker images part of a sensitive Docker Container group would perform unexpected behavior like running a command as ‘su’ or trying to connect to certain systems and/or networks.

4.4.3. Logging and monitoring of system calls

As explained, the culprit for the Dirty COW and vDSO exploits, is the madvise system call with MADV_DONTNEED flag set. In the Dirty COW exploit a root-owned setuid binary is used and in the other exploit the vDSO system is used to launch the exploit. The Linux OS allows a user to reliably monitor a set of system calls and, or file accesses through the auditd daemon. DMLA can be used to capture those type of audit logs.

The administrator then can search for the madvise system call with the MADV_DONTNEED flag set. An example of an authd configuration rule file which will logs successful madvise system calls is specified in Appendix E. Note that evidence of single madvise system call does not guarantee that the Dirty COW exploit was launched as there are valid use cases where the user, in fact, wants the OS to free the memory pages and would set the MADV_DONTNEED flag on the call. Glibc, for

Stefan Winkel, stefan@winkelsnet.com
example, might call madvise with the MADV_DONTNEED flag set on per thread basis during freeing of memory (Gorman, 2015). As the exploit is executed as part of a race condition, it is quite likely that many madvise calls would need to be made in a short amount of time. So, seeing multiple instances in the logs is probably a better indication of the exploit being executed. But even in such scenarios, further investigation will need to determine that such indicators are not False Positives (FP) as there could be other reasons for seeing multiple madvise calls in a short time span. System calls are logged by a numeric value. As there is an overlap in these values between different architectures the architecture is logged as well. In the test system used, the numeric value for the madvise system call is 28. This value is obtained by running the following command on the test system:

```bash
# for i in {1..100}; do echo $i && sudo ausyscall x86_64 $i | grep madvise ; done
```

Figure 9 ‘FP example of a madvise system call’ below displays the logging of multiple madvise system calls, identified by syscall=28. As the figure shows, there are multiple madvise system calls made in a short time span. This could lead an investigator to believe that this is an indication of a compromise. Nevertheless, none of these system calls is caused by the vDSO exploit, as the process making those calls is Java in this instance as can be seen in the picture:

Stefan Winkel, stefan@winkelsnet.com
Figure 9: FP example of a madvise system call

Even if there would not be a valid use case for such behavior, there is also the potential on False Negatives (FN) on the opposite side. With an FN, the administrator would miss the detection of Dirty Cow exploitation. This could occur when the administrator would just look for many madvise calls as an indication of exploit execution. Theoretically, only a few very few calls could trigger the exploit. This depends on when the race condition is met. Hence the indicator whether madvise library calls are made by itself is like any other indicator not sufficient by itself to determine if the exploit has been successfully launched and would require correlation of events through further investigation.

4.4.4. System Compromise through vDSO exploit

After some time, the attacker has successfully run the vDSO exploit and performed a Docker container escape. Figure 10, shows in the system logs of the ELK instance in the DMLA framework after a successful run of the vDSO exploit.

Stefan Winkel, stefan@winkelsnet.com
The reader can see that the top log entry contains the string ‘Enjoy’. This corresponds to the commands and output as were given by the attacker shown in Appendix F, Figure 16. Before running the exploit, the user was logged in as hacker@4e1effc392e8. After the exploit has been executed, the id command shows that the user id is now root, while the output of the ‘ifconfig’ shows the IP address of underlying Docker host ‘172.24.0.1’ and not the Docker container. In real world scenarios, the attacker would of course not make it so obvious that he/she is executing an exploit and most likely the exploit is modified and not show the printed statements as shown here. Instead of relying on these hardcoded strings to search through the logs, a better indication of some strange behavior is shown in the monitoring dashboard in Grafana. Figure 11 shows the CPU usage of the various containers before execution. The exploit was launched around 12:52 on 2017-06-15. The CPU usage of the exploitable Docker image ‘test4_dirtycow_run_2’ before exploitation was 0%. Figure 11 below shows that CPU usage has increased significantly to 87%. While the increase by itself does not indicate that Docker image is...

Figure 10: System compromised through vDSO exploit

Stefan Winkel, stefan@winkelsnet.com
compromised, an abnormal behavior that deviates from a normal baseline for a particular Docker image, like such a CPU spike could trigger an alert.

![Figure 11: CPU usage after vDSO exploit launch shows test4dirtycow_run_2 at 87%](image)

### 4.4.5. Machine learning and correlation of events

As indicated, one indicator alone does not guarantee that the exploit launched successfully. Correlation of indicators would increase the likelihood of a compromise occurrence, in particular with the relatively short lifespan of (potentially many) Docker containers, it might be hard to correlate such events without context. The likelihood of an administrator detecting exploits like vDSO increases significantly with accurate baselines of normal behavior of the Docker Containers. Future research using Xpack’s machine learning capabilities or commercial offerings like Prelert may prove useful in identifying trends and such baselines of Docker images and events over time. Such heuristic based logging systems might prove to be effective for long-lived Docker Images, although short-lived containers might have some additional challenges. As in general with machine learning methods like the Bayesian learning method, crucial will be to provide the system with enough observed training data to balance FPs and FNs when it comes to detecting Docker Container Escapes (Kirk, 2014).

This section showed there are quite a few steps involved in setup and establish well-parsed centralized logging from Docker Containers, but it is crucial to deploy a

Stefan Winkel, stefan@winkelsnet.com
centralized monitoring and logging framework when using Docker containers in order to detect potential malicious behavior that could have disastrous impact on company assets.

5. Conclusion

The uptake of Docker Containers has introduced many new challenges for monitoring the health of production environments, in particular with cloud-native architectures. Docker environments produce the vast amount of additional log and event data compared to traditional systems. Also, the production environments continuously change with the short lifespan of these images as the CD/CI tools keep on redeploying containers. Predictive capabilities through machine learning, together with increased automation, are frequent approaches to address these new monitoring challenges.

This article shows it is not straightforward to setup and establish a pipeline of well-parsed logs from Docker containers into a logging framework like ELK or DMLA for that matter, but the end-result is worth the effort. Being able to centralize logs and events from all the different layers and visualize them will enable forensic analyses and system administrators to correlate events and monitor their environments more efficiently.

The likelihood of another Linux kernel exploit like Dirty Cow is being discovered soon is very high. Based on the history of Linux privilege escalation bugs, such discovery will likely happen within the next six months. It will get a new CVE number and a new fancy name, but it will be just as dangerous. But a new kernel vulnerability should not be a reason to fear. With the vast amount of new logging tools and frameworks, organizations are in a good position if security practitioners and system administrators implement and use them.

Stefan Winkel, stefan@winkelsnet.com
References


Stefan Winkel, stefan@winkelsnet.com


Stefan Winkel, stefan@winkelsnet.com


Stefan Winkel, stefan@winkelsnet.com


Stefan Winkel, stefan@winkelsnet.com
Appendix Section A

dmla_config script

The dmla_config script setups up the environment to run DMLA. It installs Vagrant, VirtualBox and a few addition tools. Latest version of this file can found at https://github.com/stefanwinkel/dmla/blob/master/setup/dmla_config.bat

REM This script installs the pre-reqs for running DMLA
REM Note each phase requires a new shell to pick up the env variables from the previous command

REM Phase 0
echo "Reboot and make sure Virtualize Intel VT-x is enabled"

REM Phase 1
echo "Install Chocolatey Package Manager"

REM Phase 2
echo "It requires a new command prompt to run the remaing commands:"
echo "Install Vagrant v1.9.5 or later" & cinst vagrant -v 1.9.5 -yf
echo "Install putty v0.69 or later " & cinst putty -v 0.69 -yf
echo "Install virtualbox v5.1.22 or later " & cinst virtualbox -v 5.1.22 -yf

REM Phase 3
echo "It requires a new command prompt to run the remaing commands:"
vagrant plugin install vagrant-cachier
vagrant plugin install vagrant-multi-putty

Stefan Winkel, stefan@winkelsnet.com
Appendix section B
Building DMLA

Setting up the DMLA framework requires quite a few steps as it combines different tools and frameworks. The installation is started by running the ‘vagrant up’ command from the dmla directory. Running this command will take a while as it setups an Ubuntu Virtual image (v16.04) after which it install DMLA in a Docker container inside the Ubuntu VM.

REM Start Vagrant and launch the installation of DMLA:
C:\users\vagrant\Dekstop\dmla\vagrant up

REM To login in to the VM run the following command:
C:\users\vagrant\Dekstop\dmla\vagrant putty

Most of the operations are defined through the servers.yaml file. This file is called from the Vagrantfile in the same directory and looks something like as show in the figure below. Latest version is in the DMLA Github location.

Stefan Winkel, stefan@winkelsnet.com
After the logging portion (ELK) of the DMLA framework is up, the following manual steps need to be executed to setup the monitoring portion.

1. Log in to Grafana
   
   ```
   http://localhost:3000/dashboard/db/main-overview?orgId=1
   ```

2. Click on DataSources > Add data source
   - Name: Prometheus
   - Type: Prometheus
   - Url: `http://localhost:9090`
   - Access: Direct
   - Add

3. Click on New Dashboard and import `dmila/config/dashboards/docker-dashboard_rev5.json`

After this you should be able to view a similar Docker Dashboard in Grafana as shown in figure 8.

Stefan Winkel, stefan@winkelsnet.com
Figure 13 Docker Dashboard in Grafana
Appendix section C
Installing the vulnerable Docker image

Figure 14: Content of servers.yaml

```sh
Box: puppetlabs/ubuntu:14.04-64-nooom
GPU: 2
ip: "10.2.2.155"
name: vuln-exploit2
forward ports:
 - (guest: 8080, host: 8080)
rm: 2048
shell commands:
  - "apt-get update -y 
  - "apt-get install -y wget git"
  - "wget -qO- https://get.docker.com/ | sh"
  - "docker pull 123"
  - "sudo chmod 755 /usr/local/bin/docker-compose"

  - "git clone https://github.com/ebi/dirtycow-docker-video.git"
  - "cd /home/vagrant/dirtycow-docker-video"

  - "sudo cp /home/vagrant/vuln-exploit/docker-compose.yml /home/vagrant/dirtycow-docker-video"
  - "sudo chmod 777 /home/vagrant/dirtycow-docker-video/docker-compose.yml"
  - "sudo cat /home/vagrant/vuln-exploit/docker-compose.yml > /home/vagrant/dirtycow-docker-video/docker-compose.yml"

  - "sudo docker-compose build"
  - "echo "Logging in to the docker container...
  - "sudo docker-compose run dirtycow /bin/bash"

  - "sudo ifconfig"
  - "cd /dirtycow-video && make"
  - "sudo ifconfig"
  - "docker exec 172.18.0.2:1234 /bin/bash"
  - "wait...."
  - "sudo ifconfig"
```

Stefan Winkel, stefan@winkelsnet.com
Appendix section D
Launching the vDSO exploit

```
vagrant@dmla-v1:/dmla-v1/test_4$ cat launch_vdso_exploit.sh
#!/bin/bash
# script should be run in same VM as DMLA
# Mimics working from servers.yaml which is supposed to be run as separate VM
umask 0222
cd /home/vagrant/dmla-v1/test_4
sudo docker-compose build
echo "Logging in to the docker container"
sudo docker network create testing_net
sudo docker-compose run dirtycow /bin/bash
echo '# cd /dirtycow-vdso && make '
echo '# ./0xdeadbeef 172.18.0.2:1234 ' '
echo '# sudo ifconfig '
```

Figure 15 launch_vdso_exploit.sh
Appendix section E
Malicious operations

Figure 16: Attacker tried a failed SU command

Figure 17: Attacker trying to download exploit via Perl

Stefan Winkel, stefan@winkelsnet.com
Appendix section F
System Compromise

Figure 18: Command line output of vDSO exploit

Figure 19: Time of exploitation as show in Grafana

Stefan Winkel, stefan@winkelsnet.com
Figure 20: CPU usage before exploit launch shows test4_dirtycow_run_2 at 0%
Appendix section G
System Compromise

# This file contains the auditctl rules that are loaded
# whenever the audit daemon is started via the initscripts.
# The rules are simply the parameters that would be passed
# to auditctl.

# First rule - delete all
-D

# Increase the buffers to survive stress events.
# Make this bigger for busy systems
-b 1024

-a exit,always -S madvise
-a exit,always -S unlink -S rmdir
#-a exit,always -S stime.*
#-a exit,always -S setrlimit.*
-w /var/www -p wa
-w /etc/group -p wa
-w /etc/passwd -p wa
-w /etc/shadow -p wa
-w /etc/sudoers -p wa

# Disable adding any additional rules - note that adding *new* rules
will require a reboot
-e 2
Appendix section H
Some Docker FU

- Print Docker environment:
  
  ```bash
  sudo docker version && sudo docker info && sudo docker-compose --version
  ```

- Show Docker filesystem info:

  ```bash
  journalctl -u docker.service
  ```

- Test posting message to LogStash:

  ```bash
  curl -XPOST http://172.16.0.38:12201/gelf -p0 -d 
  '{"short_message":"Hello there", "host":"example.org", 
  "facility":"test", "_foo":"bar"}'
  ```

- Stop all Docker Containers:

  ```bash
  sudo docker stop $(sudo docker ps -a -q)
  ```

- Check if Elasticsearch has data:

  ```bash
  curl -XGET localhost:9200/_cat/indices/ -u kibana
  ```

- Delete old index:

  ```bash
  curl -XDELETE 'http://localhost:9200/filebeat-*' -u kibana
  ```

- Testfile filebeat:

  ```bash
  /usr/bin/filebeat.sh -configtest -e
  ```

- List details of Docker images and their base or intermediate images and the information of these images:

  ```bash
  sudo docker images | \
  grep $(sudo docker inspect -f "{{.Image}}" $(sudo docker ps -q) | \
  xargs -L 1 sudo docker history -q | sed "s/^/\-e /")
  ```

- Install net tools for debugging purposes:

  ```bash
  apt-get install net-tools
  ```

- Curl

  ```bash
  curl -H 'Content-Type: application/json' -XPUT 'http://172.17.0.2:9200/_template/filebeat' -d@/etc/filebeat/filebeat.template.json -u kibana
  ```

Stefan Winkel, stefan@winkelsnet.com
• Log in to ELK container: `sudo docker exec -it elkxdocker_elkx_1 /bin/bash`

• Importing the dashboards: `/usr/share/filebeat/scripts/parseimport_dashboards -es http://172.17.0.2:9200 -user elastic -pass changeme`

• Loading the kibana filebeat-* index pattern:
  `/usr/share/filebeat/scripts/import_dashboards -only-index -es http://172.17.0.2:9200 -user elastic -pass changeme`

• Check for the index in Kibana: `curl http://172.17.0.2:9200/_cat/indices?v`

• Search for filebeat data in kibana: `curl -XGET "http://172.17.0.2:9200/filebeat-*/_search?pretty" -u kibana`

Filebeat example

• `sudo docker rmi nginxfilebeat_nginx -f`

• `sudo docker build -t filebeat-nginx-example .`

• `sudo docker run -p 80:80 -it --link elkxdocker_elkx_1:elkxdocker_elkx --name filebeat-nginx-example filebeat-nginx-example`

• `sudo docker exec -it filebeat-nginx-example /bin/bash`

• Testfile filebeat: `/usr/bin/filebeat.sh -configtest -e` /usr/bin/filebeat.sh -e -v -c /etc/filebeat/filebeat.yml