



Master Thesis

Kubernetes Forensics

Christoph Lünswilken

Hannover, November 27, 2021

Examiner: Prof. Dr.-Ing. Felix Freiling, Friedrich-Alexander Universität Nürnberg-Erlangen

Advisor: Dr.-Ing. Andreas Dewald, ERNW Research GmbH

Acknowledgments

First of all, I would like and definitely need to thank a special person in my life for always being there for me, regardless of this thesis. Thank you Eileen, for never giving up on me.

The first huge kudos goes to Laura, the world's most underrated specialist for the English language. You made it work for me to write and possible for others to read this thesis. Thank you for that and all your efforts!

The second one goes to my colleagues at Syncier who helped me out wherever they could, and especially my boss Christoph, who made it possible in every way imaginable for me to get this work done besides my job. Thanks for your outstanding support!

I would also like to thank my advisor, Dr. Andreas Dewald, who made this work possible and guided me through the process of this thesis.

And not to forget my friend and fellow student Jan, who always had an open ear to my stupid questions, regarding whatever topic I didn't understand again. Thanks for listening!

And, in the end, I would like to thank all of you security researchers and developers out there in the open source community. Without all of that community effort, the world would not stand where it does today!

Abstract

The introduction of containerized applications leads to a shift in classic software architecture. Where, previously, single physical or virtual host systems have been used that contained all necessary components to run an application installed on the operating system, a more isolated approach is taken nowadays in which the single components are run in dedicated containers. With the growing usage of containers, the interests for a container orchestration software that can handle the management of distributed container environments is also rising. One of such tools is *Kubernetes*. But how does the usage of such tools affect the process of a forensic investigation? Are specific traces left by such tools that are not necessarily related to the application running in containers, or the operating system itself? This thesis covers the identification, definition and evaluation of forensic artifacts on *Kubernetes* and tries to document those. To do so, the states of a *Kubernetes* cluster before and after a performed action were compared to each other. A major objective regarding the analysis process was the automation. Traces that occur in every single run of an analysis were declared as characteristic, evaluated and probably documented as a forensic artifact to this action.

In the course of the work, the applied analysis method turned out to be applicable and generated evidences of single actions. As a result, artifacts from both *Kubernetes* and the used *container runtime containerd* were detected, described and documented in a human- and machine-readable format so that they can get contributed to the open source community. Furthermore, a preparation for forensic teams could be worked out on how to prepare specifically for incident response cases that involve a *Kubernetes* cluster and which artifacts should get saved first, based on their relevance.

Zusammenfassung

Die Einführung von containerisierten Anwendungen hat zu einem Wandel der klassischen Softwarearchitektur geführt. Wo vorher noch einzelne Systeme als physikalischer oder virtueller Host für Anwendungen dienten und dabei meist alle Komponenten auf dem selbem System installiert waren, werden diese heute in sogenannten Containern isoliert voneinander betrieben. Mit der immer stärkeren Nutzung von containerisierten Anwendungen steigt auch das Interesse am Einsatz einer Orchestrierungssoftware, um verteilte Containerumgebungen verwaltbar machen zu können. Eines dieser Tools ist *Kubernetes*. Doch wie wirkt sich der Einsatz einer solchen Software auf Prozesse im Bereich der digitalen Forensik aus? Welche speziellen Spuren fallen an, wenn entsprechende Aktionen in einer *Kubernetes*-Umgebung durchgeführt werden, die nicht notwendigerweise mit der in einem Container verpackten Software, oder dem Betriebssystem eines Hosts zu tun haben? Diese Arbeit beschäftigt sich mit der Identifizierung, Beschreibung und Bewertung forensischer Artefakte von *Kubernetes* und versucht, diese zu dokumentieren. Hierzu wurden verschiedene Aktionen in einem *Kubernetes*-cluster ausgeführt und die Zustände vor und nach der Durchführung einer Aktion miteinander verglichen. Ein Hauptaugenmerk wurde dabei auf die Automatisierung dieser Analyse gelegt. Spuren, die bei einer bestimmten Anzahl von Durchläufen einer Aktion wiederkehrend zu finden sind, werden als charakteristische Spur betrachtet, bewertet und gegebenenfalls als forensisches Artefakt dokumentiert.

Im Verlauf der Arbeit stellte sich die angewandte Methodik als anwendbar heraus und konnte Spuren einzelner Aktionen generieren. Im Ergebnis konnten Artefakte sowohl von *Kubernetes* als auch der eingesetzten Laufzeitumgebung für container, *containerd*, erfasst, erklärt und in ein maschinen- und menschenlesbares Format gebracht sowie der open-source-Community zur Verfügung gestellt werden. Ferner konnte gezeigt werden, wie sich ein Incident Response Team auf den Fall einer forensischen Analyse eines *Kubernetes*-clusters vorbereiten kann und welche Artefakte auf Basis ihrer Relevanz gesichert werden sollten.

Contents

Acknowledgments	I
Abstract	III
Zusammenfassung	IV
1. Introduction and Motivation	1
1.1. Objectives	4
1.2. Research questions	5
1.3. Thesis structure and approach	5
2. Related work	7
3. Foundations	10
3.1. Linux Namespaces	10
3.2. Container runtimes	17
3.2.1. History	17
3.2.2. OCI Image specification	17
3.2.3. containerd	21
3.3. Kubernetes	24
3.3.1. Architecture and Components	25
3.3.2. Kubernetes Objects	26
3.4. Used Tools	29
3.4.1. Virtualbox	29
3.4.2. Vagrant	29
3.4.3. Ansible	30
3.4.4. Kubectl	30
3.4.5. PowerShell	31
3.4.6. Autopsy	31

4. Concept	32
4.1. Analytic approach	32
4.2. Automation	33
4.2.1. Cluster Provisioning	33
4.2.2. Analysis with <i>PowerShell</i>	34
4.3. Analysis process	35
4.4. Example	39
4.5. Difficulties	43
5. Analysis	44
5.1. Analysis Environment	44
5.2. Scenario 1: Creation of a Pod	44
5.2.1. Description	44
5.2.2. Analysis	45
5.2.3. Results	47
5.3. Scenario 2: Run a specially crafted container	64
5.3.1. Description	64
5.3.2. Analysis	65
5.3.3. Results	66
5.4. Scenario 3: Deletion of a <i>Pod</i>	71
5.4.1. Description	71
5.4.2. Analysis	71
5.4.3. Results	72
5.5. Evaluation of container-explorer	75
6. Summary	80
7. Conclusion and Future Work	84
7.1. Future Work	84
7.2. Conclusion	86
8. Bibliography	87
A. Appendix	96
A.1. Supplements	96
A.1.1. PowerShell module: VBox4Pwsh	96
A.1.2. PowerShell module: PowerForensicator	96

A.1.3. Cluster Setup	97
A.1.4. Display-ContainerVolumes.ps1	97
A.1.5. Dockerfile: <i>etcd-dump-db</i>	98
A.1.6. Dockerfile: <i>container-explorer</i>	98
A.1.7. Kubernetes artifacts	98
A.2. OCI Image specification examples	99
A.2.1. Example image configuration	99
A.2.2. Example image index	101
A.3. Example analysis scenario	102
A.3.1. Characteristic evidences of master node	102
A.3.2. Characteristic evidences of worker node	102
A.3.3. Selected fileobject of <i>idiff</i> file	104
A.3.4. Selected entry of fileobject	105
A.4. Analysis environment	108
A.5. Scenario 01: Deployment of a Pod	108
A.5.1. Analysis.ps1	109
A.5.2. simplePod.yaml	109
A.5.3. Result files - Ubuntu	109
A.5.4. Result files - Windows	110
A.5.5. Exported files	111
A.6. Scenario 02: Deploy crafted Container	111
A.6.1. Analysis.ps1	111
A.6.2. Dockerfile	112
A.6.3. craftedPod.yaml	112
A.6.4. DownloadFiles.sh	112
A.6.5. Result files - Ubuntu	113
A.6.6. Result files - Windows	113
A.6.7. Exported files	114
A.7. Scenario 3: Deletion of a <i>Pod</i>	115
A.7.1. Analysis.ps1	115
A.7.2. CraftedPod.yaml	115
A.7.3. Result files - Ubuntu	115
A.7.4. Result files - Windows	116
A.7.5. Exported files	117

Statutory Declaration	118
------------------------------	------------

List of Figures

3.1.	Content of the <code>cgroup</code> directory	15
3.2.	CPU controls in the newly created namespace <code>TestApplication</code> . . .	16
3.3.	Architecture of <i>containerd</i>	22
3.4.	Data flow of an image pull operation in <i>containerd</i>	23
3.5.	The components of a Kubernetes cluster	25
5.1.	Key-Value pair of index 14 on page 26 of <code>metadata.db</code>	52
5.2.	Contents of page 24 of <code>metadata.db</code>	53
5.3.	Image information from <i>Docker</i> image registry	57
5.4.	Volumes mounted by <i>etcd</i> container	59
5.5.	List of buckets in <i>etcd</i> database	60
5.6.	Content of mounted volume	63
5.7.	Pushing an image in <i>Docker</i>	65
5.8.	New checksum after extraction of a layer	68
5.9.	Manually downloaded files shown in <i>Autopsy</i>	69

List of Tables

- 3.1. Properties of an image configuration as specified by the *OCI* 20
- 4.1. Commandlet overview of the *Vbox4Pwsh* module 35
- 4.2. Meaning of the reported file records 37
- 4.3. Expected characteristic evidences 41
- 5.1. Original identifiers from last round of analysis 51
- 5.2. Overview of custom files 64
- A.6. Overview of software components used for the analysis 108

List of Listings

3.1. Example manifest of an <i>OCI</i> image	19
4.1. Files created for <i>Init</i> state	39
4.2. File modifications performed during the <i>Action</i> phase	40
4.3. Selected characteristic evidences of worker node	42
5.1. Specification of the <i>Pod</i> resource	45
5.2. Contents of the <code>Analysis.ps1</code> file of the scenario: Deploy pod	46
5.3. Overview of replaced strings	47
5.4. Characteristic file system mount evidences from scenario 1	48
5.5. Perform analysis on <code>metadata.db</code>	50
5.6. Content of the file <code>status</code>	54
5.7. Content of the detected file	55
5.8. Download of files during the <i>Action</i> -phase	66
5.9. Characteristic file system evidences detected on master node	67
5.10. Logged entry in <code>syslog</code>	70
5.11. Command that deletes the <i>Pod</i>	72
A.1. Example image configuration	100
A.2. Example image index	101
A.3. Characteristic evidences of master node in the example scenario	102
A.4. Characteristic evidences of worker node in the example scenario	103
A.5. Content of selected <code>idiff</code> file showing the file object <code>delete.me</code> . . .	105
A.6. Content of selected <code>idiff</code> file showing the file object <code>IhaveBeen.re</code> named	107

INTRODUCTION AND MOTIVATION

With the introduction and usage of containerized applications, a shift from complex infrastructures to small, scalable environments can be observed (see Hasselbring and Steinacker, 2017). Complex web applications that previously have been deployed to single server instances and therefore required a high maintenance effort are now migrated to microservice infrastructures making use of container technologies like *Docker* (Docker, 2021a). Providing additional features to manage a large amount of such containers, *Kubernetes* (Kubernetes, 2021i) as an orchestration environment has become popular over the last years. *Kubernetes* is an open source tool maintained by the *Cloud Native Computing Foundation (CNCF)* (CNCF, 2021), designed to scale, manage and automate deployments of containerized applications. Several different kinds of deployments as well as *Kubernetes* resources are distributed to worker nodes which form a *Kubernetes* cluster. This *Kubernetes* cluster can be referred to as the installation of a *Kubernetes* environment using different architecture models, e.g. single-node or multi-node installations. When it comes to security for those *Kubernetes* clusters, one has various sets of configurations to enhance security, which mostly apply to the fields of detection and prevention in the context of IT security. But what if all those prevention methods are not sufficient? This is where digital forensics and incident response (DFIR) enters the picture, whose main goal is to react to security incidents in the shortest time possible by detecting and examining affected artifacts in order to provide a systematic processing of the security incident. The first question that an incident response team might come up with is: What are the most relevant forensic artifacts that need to be collected in order to make a statement about the ongoing incident? And what are possible ways for collecting those? In a more classic environment, where a web application is hosted on a single-server instance together with all

its components, that question could be answered easily because most of the necessary artifacts are located on that particular server. A memory dump, for example, could be taken by the hypervisor software or a snapshot of the whole disk image could be exported. In order to perform a live analysis, the incident response team would only need a single access to that particular server because all related artifacts could be monitored from there. This highly differs when it comes to *Kubernetes*. Although one might argue that in *Kubernetes* clusters the used components just split to different containers and therefore the forensic process would be nearly the same as in a classic environment, the way the web application has been deployed brings in some changes to the process of artifact collection. This is also affected by the change in the way containers are handled within the architecture itself. The administration of containers follows the principle of being ephemeral, meaning that the configuration for a container or even whole *Kubernetes* clusters is stored as code in configuration files maintained in version control systems. This allows an administrator to simply redeploy a container in case of a failure, instead of troubleshooting the error until the running container is working again as expected. In case of a security incident, where a whole cluster might be affected, the administrators could just decide to take the cluster down and build it from scratch using the infrastructure or configuration as code approach, instead of analyzing the breach together with an incident response team. All possible log files from the deployed resources within that cluster would be gone, given they are not actively fetched by a centralized logging system (see LeClaire, 2020, pp. 28–30).

Kubernetes itself uses specific top-level components that are necessary to manage the underlying cluster and all of the deployed parts within. For example, there is the *Kubernetes* master node which hosts components like the *kube-api-server*, *controller manager*, *scheduler* and the internal database called *etcd* (Kubernetes, 2021h). This set of components is used to control internal cluster traffic as well as distributing workloads to the *Kubernetes* worker nodes. Therefore, it is important to know which *Kubernetes* components contain forensic artifacts that are crucial to analyze besides those that reside on containers that host the web application. A basic understanding and knowledge of artifacts that are specific to *Kubernetes* is necessary in order to perform a forensically sound investigation process. If one focuses only on collecting artifacts within running containers, valuable information could be missed out by not collecting artifacts from the *Kubernetes* components as well.

The shift to microservice architectures deployed in *Kubernetes* clusters also brings in new kinds of threats that differ from classic web application or database-specific attacks. Often seen examples are attacks that aim to steal compute resources within a cluster en-

vironment. A containerized crypto-mining application could be placed into the cluster by attackers which exploits the provided resources and thus drives up costs for the owner (see RedLock, 2018). If an incident response team has to analyze such kind of an attack, it is even more crucial to have detailed knowledge on where to find *Kubernetes*-specific forensic artifacts that might tell when, from where and by whom such a container has been deployed. A main point to start the investigation at would be the examination of the *Kubernetes* audit logs, if any are given. As stated in the official documentation (Kubernetes, 2021b), the audit records provide answers to a large amount of questions: What happened, where and when did it happen, who was the initiator, from where was it initiated and where was it going? The level of details of such a record is based on the configured audit policy of a cluster. This could cause a problem for incident response teams. If an attack has already happened or is still going on, but the log level defined within that policy is not detailed enough, information necessary for tackling the attack would be missing. In this case, a team would need to know if there are other *Kubernetes*-related forensic artifacts available that could help out in such a situation. Furthermore, the audit policy allows to define a log level granularly for each type of resource within a cluster. One could therefore define to log all possible data for the resource *pods* but explicitly log nothing for the *configmap* resource. Given that this resource was the goal of an attack and changes have been made, what kind of information is missing? And what other possible sources of information are available to fill the gap of this missing information?

1.1. Objectives

One key objective of this work is the definition of files and information that are useful in a forensic investigation process involving *Kubernetes* cluster components. An overview of all components included in a basic *Kubernetes* installation will be given. The identified forensic artifacts will be brought into a standardized and machine-readable format so that they form a knowledge base and can be used in other tools. Garfinkel (2012b) worked out a schema called *DFXML*, which aims to structure forensic information in order to be exchangeable. The main focus is on standardizing the output of forensic tools. This schema as well as further tools can be found in the actively maintained GitHub-Repository (Garfinkel, 2012a). The repository further includes several scripted use cases of that schema, like the `idifference2.py` script which uses the *sleuthkit* (Carrier, 2011) tool `fiwalk` that contains the functionality of parsing disk image files using the *sleuthkit* library and outputs the results in a DFXML format. Using `fiwalk` as a base, `idifference2.py` will be used to analyze the differences between states of a disk image in order to search for characteristic evidences. The structure of forensic information itself could be aligned with another already existing open source project called *ForensicArtifacts* (Castle and Metz, 2014) and a resulting contribution to that project, providing new artifacts, is possible. Furthermore, the robustness of these forensic artifacts needs to be proven by performing tasks in the cluster which will be observed in the best possible way, for example, by taking snapshots before and after an action and then comparing the states.

It is necessary to define beforehand which components are affected by an action at all. This includes all kinds of administrative tasks as well as dedicated attacks. A sample web application could be hosted and attacked by most common attack scenarios. Instead of taking a look into forensic artifacts that belong to the hosting service (like *Apache*, *NGINX* etc.), a closer look at the *Kubernetes* components will be taken. An artifact which occurs every time a specific action is performed within the cluster can be categorized by that and will be documented within the knowledge base.

A further objective is the definition of ways in which a forensic artifact can be collected from a *Kubernetes* cluster. If any requirements are necessary in order to have those artifacts collectable, those will also be documented in the knowledge base.

1.2. Research questions

As *Kubernetes* provides a tremendous amount of functionalities that are configurable through commands, many actions take place inside a *Kubernetes* cluster or, more specifically, between the *Kubernetes* components. This leads to the main question that will be answered by this work: Do the actions mentioned in section 1.1 produce characteristic evidences in a *Kubernetes* environment? Do those evidences differ from the audit logs and can they be used in addition to or even instead of those audit logs? In case characteristic evidences are given, can they be documented and classified as forensic artifacts using a standardized notation model and evaluated whether they are useful for threat detection or incident response processes?

Furthermore, the extraction process of forensic artifacts from *Kubernetes* clusters will be looked at. Are already existing tools sufficient to extract and inspect forensic artifacts from a *Kubernetes* cluster or do they lack functionalities that would be necessary in order to fulfill requirements of a forensically sound artifact processing? If not, could an own tool be developed in the course of this thesis that can close possible gaps? And what precautions does an incident response team need to take before starting an investigation process? *Kubernetes* clusters can be deployed in different ways and on different platforms. So, does the way the cluster is deployed affect what kind of forensic artifact a *Kubernetes* component produces?

1.3. Thesis structure and approach

Up to this point, Chapter 1 has given an overview about the motivation behind this thesis and an insight as to how the identification of artifacts could be achieved. Furthermore, the research questions have been phrased and will be tackled within the upcoming sections.

Chapter 2 will discuss approaches taken by other researchers and what the base of an analytic approach could look like. Besides that, a set of tools is mentioned that could be used within this work to perform forensic tasks related to containerized environments.

Chapter 3 will deliver the theoretical foundations needed to understand the ways of how containers work on an operating system. An insight into *Linux namespaces* will be given in Section 3.1, followed by an explanation of the standards of containerization as defined by the *OCI*. The section will also cover the container runtime that is used within the further work. A section that introduces *Kubernetes* and its core components will bring the

standards of container runtimes into context. The chapter is concluded with an overview of further tools that are being used for the analysis.

Chapter 5 will explain the analytic approach taken in this thesis in detail. As the approach is heavily relying on automation, the topics of automated cluster provisioning and the automation of the analysis process itself will also be covered. An example for the analysis process will be given that aims to prove the forensic correctness of identified characteristic evidences by performing an example scenario that produces expected results. Those results as well as encountered problems will be discussed at the end of that chapter.

Chapter 5 contains the documentation of the performed analysis. In sum there were three analysis scenarios developed and analysed, each covering a unique action while being related to each other. In addition to that, the tool *container-explorer* will be evaluated in that section.

The results from the performed analysis will be picked up and evaluated in Chapter 6. Furthermore, it will be discussed whether the research questions were answered or not.

A prospect of possible enhancements and further analysis approaches based on the results of this work will be given in the concluding Chapter 7.

RELATED WORK

Speaking of cloud-specific forensic investigation processes, there are already some tools available that aim to automate the forensic investigation process by collecting artifacts or ingesting those. Many of those tools are written, maintained and distributed by *Google*. The *cloud-forensics-utils* (Google, 2020) repository in *Github*, for example, contains several scripts used by forensic investigators to perform diverse actions. Besides that, there is the *Google Rapid Response (GRR)* (Google, 2013) platform which provides a client-server-infrastructure that has the ability to perform live forensic tasks within cloud environments, including *Kubernetes* clusters. The tool itself includes a list of well-known forensic artifacts that apply to specific systems where the *GRR* agent has been installed. Those can automatically get collected via a GUI by the forensic analysts. Furthermore, it is possible to create a distributed collection of artifacts across a fleet of systems. This list of artifacts has been provided to the open source community and is now further developed and maintained within the *ForensicArtifacts-Repository* (Castle and Metz, 2014). So far it does not include *Kubernetes*-specific artifacts or other information that would apply to *Kubernetes* components. The tools mentioned could be used in this work to automate the process of extracting forensic information. Enhancing the list of forensic artifacts might be possible so that the artifacts can be selected within the GUI of *GRR*. Possible results of this work could also be distributed to the community or to the repositories in *Git*Hub respectively. In case neither of the aforementioned tools are sufficient to extract the forensic information discovered through this work, either an own tool has to be developed that is capable of closing this gap, or enhancements to the existing tools need to be made, if possible.

Turbinia (Google, 2015) and *Timesketch* (Google, 2014) are two other tools developed by

Google. *Turbinia* is more like an environment that uses clients, a server and worker nodes to distribute forensic tasks with a high workload, e.g. the ingestion of huge raw data files that contain a forensic copy of a hard disk drive or memory snapshots, etc. These so-called jobs are issued by the clients and processed by the worker nodes. *Timesketch*, on the other hand, is used to visualize huge timelines consisting of timestamps from different sources. One tool that is often used to create such a timeline is *plaso* (log2timeline, 2014). Another tool that is capable of orchestrating the process of forensic investigation, including the collection of forensic artifacts by using subprograms, is *DFTimewolf* (log2timeline, 2016) which is also maintained by the *log2timeline* project. The orchestration is described in so-called recipes. Some of those already include actions taken in *Google*'s and *Amazon*'s cloud products as well as different interactions with *Google*'s tools mentioned before.

A more theoretical approach when it comes to secure configuration of systems is delivered within the benchmarks from the *Center of Internet Security (CIS)* (CIS, 2020). Among various types of other systems, a benchmark for *Kubernetes* infrastructures has been disclosed. This benchmark collects some of the best practices as controls, providing information on the control itself, why it should be implemented, what it will enhance and how to audit if the control has been applied to an infrastructure. While this collection of controls provides a comprehensive overview of security settings in *Kubernetes*, it does not include information on forensic artifacts themselves. Again, all measures aim to harden the environment and therefore prevent security breaches instead of focusing on forensic analysis affecting the *Kubernetes* components. The benchmark will be used to see if results from this work correlate with the defined best practices.

As mentioned above, digital evidence is often found in the log files that are provided by the applications and their components themselves that have been deployed in a *Kubernetes* cluster. The work from Riadi et al. (2020) contains a research on how to access these artifacts using *GRR*. A DoS attack has been executed against a *Kubernetes* environment which hosted a web service. The corresponding containerized web server provides the necessary log files that are used to analyze the attack. This work may get enhanced by taking a look at the states of the *Kubernetes* components before and after a DoS attack, having the tool *GRR* deployed to the cluster.

In the area of analytical methods, Garfinkel et al. (2012) describe an approach of differential strategies applied in forensic analysis. Given that a system contains both volatile and persistent data, a snapshot is taken at two different stages. These two snapshots will be compared to each other so that the differences are pointed out. The identified differences

will give information on what has changed within the time slot between those two stages. The next goal then is to find out characteristic changes that are reliably measured to a specific action that happens between these two stages. In addition to that, the work of Dewald (2015) discusses the notions of digital evidences and introduces the terminology of characteristic evidences, also discussed in Dewald (2012). Searching for characteristic evidences within *Kubernetes* components by following the implementation model of a forensic fingerprint generator introduced by Kalber et al. (2013) will be the main part of this work. As shortly described in Section 1.1, a *Kubernetes* cluster will be virtualized using *Virtualbox*. This allows to programmatically extract forensic images of the given instance before and after an action in order to point out differences between those two states. The extracted raw images will be brought into a comparable format, using the tool *fiwalk* (see Garfinkel, 2009) which produces a machine-readable *DFXML* format. As mentioned above, this format as well as the necessary tools are available in the *Github* repository *DFXML* (Garfinkel, 2012a).

FOUNDATIONS

3.1. Linux Namespaces

The word *container* is a well-known term used in the area of the shipping industry, where it describes an often rectangular-shaped object that contains goods of all various kinds. Using containers makes it easier to transport goods from one location to another due to the standardization that applies to such containers, like the specified size and volume capacity. Plus, the goods are shipped in an isolated environment provided by the container, thus making the goods resistant to external impacts such as rain, storm and all other influences during the shipping process. This explanation of a container being an isolated, standardized and shippable unit is now also used within the area of software development, where it describes the method of wrapping software with its dependencies, binaries etc. into a dedicated environment as well as isolating resources used by the specific software so that it can run in a standalone way from any operating system. This containerization needs to be provided on operating system level which needs to support the creation of dedicated resources where the software components can run in. This said, a container should fulfill the requirements of running on a single host inside an isolated environment and consisting of a set of processes to make the shipped software in the container run (Grunert, 2019).

Such features are provided by namespaces (see Linux man-pages project, 2021g), which are part of the *Linux*-kernel (Linux Kernel Organization, 2021b). Namespaces create an isolation layer by abstracting global system resources into dedicated areas, providing processes with an isolated and dedicated environment. Resources in that namespace can only

be shared with other processes running in that namespace. As of today, the stable kernel release in version 5.13.5 ships with eight implemented namespaces: `cgroup`, `IPC`, `Network`, `Mount`, `pid`, `Time`, `User` and `UTS`. The namespaces are controlled via the namespaces API which includes several system calls. The `clone()` system call creates a new child process and makes this new process a member of the created namespace while controlling precisely which parts of the execution context should be shared (Linux man-pages project, 2021d). Similar to `clone()` is the `fork()` system call. This call duplicates the calling process and shares the execution context while running in a separated memory space. The caller's parent process ID will stay the same for the newly created process (Linux man-pages project, 2021e). Another call that is capable of creating new namespaces and moving a process into it is `unshare()`. The calling process gets extracted from its current execution context and will be in control over the new execution context inside the created namespace. If the calling process does not specify a program that should get started in the new namespace, then the configured default shell will get executed (Linux man-pages project, 2021n). To finally enter a namespace, the system call `setns()` can be used. This call allows the calling process to enter an already existing namespace, thus sharing the resources amongst the other processes within that namespace (Linux man-pages project, 2021k). Another call similar to that is `nsenter()`, which executes a new program within a given namespace. A shell binary could, for example, be started in a particular namespace by invoking `/bin/sh` using `nsenter()`. Each process having a namespace assigned will receive a corresponding entry within the `/proc` directory of *Linux*. They are accessible via the pseudo sub-symbolic links (Linux man-pages project, 2021g,j).

The namespace `mnt` allows an isolation of processes within a namespace. When not in a namespace, a process can see the complete list of mount points that the operating system currently has configured. If a process is running within a namespace, this list of mount points can be controlled. In case a `mnt` namespace gets created via `clone()` system call, the cloned process will have the same mount list or mount namespace as the parent's one. If created via `unshare`, the calling process keeps a version of the previous mount-namespace, while the child process will get a new namespace. So, when a new `mnt`-namespaces gets created, the process to which that namespace has been assigned to will only see files within its own namespace. Using this functionality via the `unshare()` system call, the flag `CLONE_NEWNS` is required which also automatically implies `CLONE_FS`, meaning that the calling process will not share its root directory any longer. Additionally, the attributes set by `umask` will not be set for the child process

namespace (Linux man-pages project, 2021f,n).

The `pid`-namespaces aim to provide isolated environments for processes. The abbreviation `pid` stands for *Process ID* which identifies a process running in an operating system by a unique identifier, mostly displayed in digits. In a regular *Linux*-based operating system, the process with the ID 1 is the `init` or `systemd` process which gets started during the boot sequence and invokes further processes as children. That means the `systemd` process is at top level of a process tree in *Linux* systems (Linux man-pages project, 2021l) and no other process can get the process ID of 1 assigned. Within `pid`-namespaces, though, there are no already existing process trees. The first process that gets created in such a namespace will get assigned the process ID 1. The system calls `clone()` and `unshare()` both share the same flag called `CLONE_NEWPID` for creating a new `pid`-namespace. When using `clone()`, the parent process will keep its process ID while the cloned process within the newly created `pid`-namespace will have the ID 1. The `unshare()` system call will assign the PID 1 to the first process to spawn within the new namespace, most often the configured default shell. If another process's parent within the new namespace gets terminated, the process with ID 1 will become the new parent of that now orphaned process. This scenario applies to both `unshare()` and `clone()`. Furthermore, it is possible to create a `pid`-namespace within another `pid`-namespace.

The calling process will, for example, get the process ID 2 assigned, while the cloned or forked process within the newly created namespace will get the PID of 1 becoming more or less a child `pid`-namespace. If the top level parent process of those created (and nested) namespaces gets killed, all subsequent namespaces will be removed as well as the child processes will be stopped. In the end, the concept of a `pid`-namespace allows to create an isolated environment for newly spawned processes so that they can run independently from other resources (Linux man-pages project, 2021d,i,n).

Another namespace called `Unix Timesharing System` (UTS) introduces the concept of separating the host and domain name from the operating system. Both `unshare()` and `clone()` use the flag `CLONE_NEWUTS` in order to specify that a new `uts` namespace shall get created. The properties of the calling process's namespace will be copied over to the newly created one and can get altered within this new namespace. Changes done to the properties within the new `uts` namespace will not affect the properties of the calling process or the operating system itself (Linux man-pages project, 2021p).

The `Interprocess communication` (`ipc`) namespace handles the communica-

tion between multiple processes which might consist of `ipc`-objects like `System-V` Objects (Linux man-pages project, 2020c) or `POSIX` message queue (Linux man-pages project, 2020b), for example. In that, the processes within an `ipc`-namespace are allowed to use a shared memory while each namespace has its own `ipc`-object identifiers and `POSIX` message queue file system. This means that the `/proc` interfaces in `/proc/sys/fs/mqueue`, `/proc/sys/kernel` and `/proc/sysvipc` are distinct from each namespace. In order to create this namespace, the flag `CLONE_NEWIPC` needs to be set and is used by both `clone()` and `unshare()`. When used with `unshare()`, the flag `CLONE_SYSVSEM` will automatically be set, which is not the case for `clone()` (Linux man-pages project, 2019).

Given that host and domain names can be changed within namespaces, opens up the theory of also sharing networking resources through namespaces. This is where the `net`-namespace comes into play. The network-related resources of the operating system, such as physical and/or virtual network devices, protocol stacks, firewall rules, routing tables, etc., can be shared within isolated environments through the `net`-namespace. There are some restrictions though, e.g. that a physical network device can only exist in one single `net`-namespace. Once the namespace gets removed, the device will move back to the namespace where it originated from. Besides the physical network devices, virtual ones can also get created and be used in several scenarios. The virtual network devices provide functionalities to communicate between different namespaces and with the public internet. In that, they can get configured as a network bridge to where one end gets plugged into another (physical) network device that can access the internet while the other end acts as a network interface in the respective `net`-namespace. Technically speaking, the virtual network devices are `Virtual Ethernet device (veth)` (Linux man-pages project, 2021q) pairs in the operating system. Those various kinds of connectivities would allow for the creation of software-defined networks (SDN), which is an important feature used in container runtimes. When a `net`-namespace gets created, a default loopback interface will be assigned to it. In order to create such a namespace, the flag `CLONE_NEWNET` must be used either in `unshare()` or `clone()` system call (Grunert, 2019; Linux man-pages project, 2021h).

Another topic covered by a dedicated namespace is the user management. The `user`-namespace allows to separate the user and group ranges from the host so that no overlap is given between the host's and the namespaces user ID (`uid`) or group ID (`gid`) usage. The permission level of a process inside the namespace can be different from outside the namespace. This means that a process inside the namespace can get the user ID 0

assigned and therefore run with root privileges, while the calling process from outside the namespace might have the user ID 1000 and runs as an unprivileged process. It is further possible to have nested `user-namespaces`. There is an initial root-namespace that does not contain a parent `user-namespace`, but all child namespaces can have one single or multiple child namespaces themselves. Again, this namespace can be created by using either `clone()` or `unshare()` system calls which will define the mapping between the parent `user-namespace` and the new one based on the calling process. The process within the new namespace will also get security bits assigned which define the capabilities (Linux man-pages project, 2021a, see) that the process is able to use. Even though a large set of capabilities, which allow for privileged operations, can be set, not all capabilities can be used outside the initial root `user-namespace`. Such are the `CAP_SYS_TIME` capability which governs operations to change the system time and date, or `CAP_MKNOD` which contains the privileges to create devices. The `CAP_SYS_MODULE` which allows to load kernel modules is also affected by this restriction and therefore not available in child or nested `user-namespaces` (Linux man-pages project, 2021o).

The final namespace left is taking care of the system resources itself. With the introduction of `cgroup-namespaces`, the *Linux* kernel got added the feature of virtualizing the so-called *Linux Control Groups* (*cgroups*). A `cgroup` describes a set of processes for which the access and usage to specific system resources can be managed and monitored by predefined controllers. One can granularly set the CPU time that a `cgroup` is allowed to consume, or directly bind an arbitrary number of CPU cores directly to the group. Furthermore, the memory controller provides limitation of memory usage. The process memory itself can be limited as well as the kernel memory consumption by the `cgroup`. The swap usage is also manageable. The device controller can be used to control whether a process may create new devices or is allowed to perform read/write operations on already existing devices. Access to block devices can also get managed via the `blkio` controller. The `pids` controller limits the number of total processes that are allowed to be created within a control group. For the networking side, the `net_prio` controller is used to specify networking priorities per interface in a `cgroup`. Networking classes can be configured by using the `net_cls` controller, which takes class-identifiers in order to use them in firewall rules that apply for the outgoing network traffic of that `cgroup` (Linux man-pages project, 2021b). The concept of having `cgroups` is adopted by the `cgroup-namespaces`, which allow to create namespaces using the already known `unshare()` and `clone()` system calls. To create such a namespace, the flag `CLONE_NEWCGROUP` must be used. The `cgroups` use a pseudo file system called `cgroupfs` which is usu-

ally mounted at `/sys/fs/cgroup` in *Linux* operating systems. The aforementioned controllers are mounted in that directory.

```
vagrant@k8s-m-1:~$ sudo ls -la /sys/fs/cgroup/
total 0
drwxr-xr-x 15 root root 380 Sep 28 12:33 .
drwxr-xr-x 10 root root  0 Sep 28 12:32 ..
dr-xr-xr-x  6 root root  0 Sep 28 12:33 blkio
lrwxrwxrwx  1 root root 11 Sep 28 12:33 cpu -> cpu,cpuacct
lrwxrwxrwx  1 root root 11 Sep 28 12:33 cpuacct -> cpu,cpuacct
dr-xr-xr-x  6 root root  0 Sep 28 12:33 cpu,cpuacct
dr-xr-xr-x 21 root root  0 Sep 28 12:33 cpuset
dr-xr-xr-x  6 root root  0 Sep 28 12:33 devices
dr-xr-xr-x 21 root root  0 Sep 28 12:33 freezer
dr-xr-xr-x 21 root root  0 Sep 28 12:33 hugetlb
dr-xr-xr-x  6 root root  0 Sep 28 12:33 memory
lrwxrwxrwx  1 root root 16 Sep 28 12:33 net_cls -> net_cls,net_prio
dr-xr-xr-x 21 root root  0 Sep 28 12:33 net_cls,net_prio
lrwxrwxrwx  1 root root 16 Sep 28 12:33 net_prio -> net_cls,net_prio
dr-xr-xr-x 21 root root  0 Sep 28 12:33 perf_event
dr-xr-xr-x  6 root root  0 Sep 28 12:33 pids
dr-xr-xr-x  2 root root  0 Sep 28 12:33 rdma
dr-xr-xr-x  6 root root  0 Sep 28 12:33 systemd
dr-xr-xr-x  6 root root  0 Sep 28 12:35 unified
```

Figure 3.1.: Content of the `cgroup` directory

In order to create a namespace, a new directory needs to get created in a controller mount directory. This sub-directory will automatically get the possible configuration files including default values mounted. In the following example, the folder `TestApplication` has been created within the `cpu` controller mount.

It is now possible to configure several characteristics of CPU usage. To bind a process to this `cgroup`, the PID of that process has to be added to the file `/sys/fs/cgroup/cpu/TestApplication/cgroup.procs`. The configurations will then also be automatically added to the corresponding entries within the file `/proc/{PID}/cgroup` (Linux man-pages project, 2020a) (Grunert, 2019).

Each of the namespaces described above brings important features when it comes to the isolation of system resources. The true power behind that is revealed when multiple namespaces are composed together. In a simple example, isolated `pid`-namespaces could share the same network while working on the same file system provided via `mnt`-namespaces. Composed namespaces define the foundation of containers, which are managed through container runtimes like *runc*, *containerd* or *Docker*, which enhance the

```
vagrant@k8s-m-1:~$ sudo mkdir /sys/fs/cgroup/cpu/TestApplication
vagrant@k8s-m-1:~$ sudo ls -la /sys/fs/cgroup/cpu/TestApplication
total 0
drwxr-xr-x 2 root root 0 Sep 30 18:27 .
dr-xr-xr-x 8 root root 0 Sep 28 12:33 ..
-rw-r--r-- 1 root root 0 Sep 30 18:27 cgroup.clone_children
-rw-r--r-- 1 root root 0 Sep 30 18:27 cgroup.procs
-r--r--r-- 1 root root 0 Sep 30 18:27 cpuacct.stat
-rw-r--r-- 1 root root 0 Sep 30 18:27 cpuacct.usage
-r--r--r-- 1 root root 0 Sep 30 18:27 cpuacct.usage_all
-r--r--r-- 1 root root 0 Sep 30 18:27 cpuacct.usage_percpu
-r--r--r-- 1 root root 0 Sep 30 18:27 cpuacct.usage_percpu_sys
-r--r--r-- 1 root root 0 Sep 30 18:27 cpuacct.usage_percpu_user
-r--r--r-- 1 root root 0 Sep 30 18:27 cpuacct.usage_sys
-r--r--r-- 1 root root 0 Sep 30 18:27 cpuacct.usage_user
-rw-r--r-- 1 root root 0 Sep 30 18:27 cpu.cfs_period_us
-rw-r--r-- 1 root root 0 Sep 30 18:27 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 Sep 30 18:27 cpu.shares
-r--r--r-- 1 root root 0 Sep 30 18:27 cpu.stat
-rw-r--r-- 1 root root 0 Sep 30 18:27 cpu.uclamp.max
-rw-r--r-- 1 root root 0 Sep 30 18:27 cpu.uclamp.min
-rw-r--r-- 1 root root 0 Sep 30 18:27 notify_on_release
-rw-r--r-- 1 root root 0 Sep 30 18:27 tasks
vagrant@k8s-m-1:~$ |
```

Figure 3.2.: CPU controls in the newly created namespace TestApplication

runtime with additional features. Container runtimes use the *Linux* namespaces as the underlying technology for containerization (Grunert, 2019).

3.2. Container runtimes

3.2.1. History

To compose the functionalities of *Linux Namespaces* and use them at scale, *container runtimes* come into play. They act as a layer between the plain *Linux* kernel functions and the developer or other software components. One of the first introduced *container runtimes* was *Docker* (Docker, 2021a). It provided a first try of a standardization of containers by streamlining the image formats, build processes, management and the actual running of containers. This was achieved by providing a simple command line tool that translated commands like `docker run` into the underlying system kernel so that the necessary components would get composed and created. In order to standardize description and storage format of containers between different vendors, *Docker*, *Google* and *CoreOS* formed the *Open Container Initiative* (OCI) project (OCI, 2021), whose purpose is to develop and maintain standards for containers. Currently, the *Runtime Specification* (*runtime-spec*) (OCI, 2015a) and the *Image Specification* (*image-spec*) (OCI, 2016e) are described by the *OCI*. The *image-spec* describes what information must be included within a standardized container image file to be used in a compliant way. It will contain information about the included files organized as a layer, the commands to run and other specific information (see Section 3.2.2). The developers of *Docker* decided to donate the runtime implementation to *OCI* which is now maintaining it under the project *runc* (OCI, 2015c), forming a low-level *container runtime* implementation. Since then, various other *container runtimes* were developed, each serving a different purpose. They are sorted into low-level and high-level *container runtimes*. The low-level ones tend to be integrated into more high-level tools that use APIs to run them, whereas high-level *container runtimes* are often used directly by users to work with containers and resources. One common use case is a user interacting with a high-level *container runtime* which makes use of a low-level *container runtime*.

3.2.2. OCI Image specification

Every container is built upon an image that needs to meet the requirements specified by the *OCI*. This image contains four components (OCI, 2016a). The image manifest aims to streamline the configuration of an image thus enabling it to be addressable by its contents. It stores metadata about all components of the image itself as properties like file system

layers, the configuration object and additional information such as annotations. The version of the manifest is set by the property `schemaVersion` and ensures compatibility with *container runtimes*. A configuration object must be referenced within the `config` property and the layers which build the file systems are referenced within the `layers` property. The values used for referencing within both properties must be digests, which are part of the content descriptors also specified by the *OCI* (OCI, 2016c). To calculate such a digest, the byte content of an object will be hashed using either SHA256 or the SHA512 algorithm. This ensures that a content gets addressable and can be identified. The digest is therefore also used to validate the contents of the image to make sure that nothing got modified. The last property of a manifest is the *annotation*, which includes a map of custom string values. An example *manifest* may look like the following:

```

1 {
2   "schemaVersion": 2,
3   "config": {
4     "mediaType": "application/vnd.oci.image.config.v1+json",
5     "size": 7023,
6     "digest": "sha256:b5b2b2c507a0944348e0303114d8d93aaaa081732b86451j
   ↪ d9bce1f432a537bc7"
7   },
8   "layers": [
9     {
10      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
11      "size": 32654,
12      "digest": "sha256:9834876dcfb05cb167a5c24953eba58c4ac89b1adf57fj
   ↪ 28f2f9d09af107ee8f0"
13    },
14    {
15      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
16      "size": 16724,
17      "digest": "sha256:3c3a4604a545cdc127456d94e421cd355bca5b528f4a9j
   ↪ c1905b15da2eb4a4c6b"
18    },
19    {
20      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
21      "size": 73109,
22      "digest": "sha256:ec4b8955958665577945c89419d1af06b5f7636b4ac3dj
   ↪ a7f12184802ad867736"
23    }
24  ],

```

```
25  "annotations": {  
26    "com.example.key1": "value1",  
27    "com.example.key2": "value2"  
28  }  
29 }
```

Listing 3.1.: Example manifest of an *OCI* image. Taken from (OCI, 2016g)

The second component of an *OCI*-compliant image is the configuration which contains several properties in a JSON-structured file. They tell the corresponding *container runtime* how to execute the image and contain further information about the changes in the file system.

Property	Mandatory	Description
created	No	Creation timestamp of the image.
author	No	Information about the author of an image.
architecture	Yes	The CPU architecture which the binaries in this image are built to run on.
os	Yes	The name of the operating system which the image is built to run on.
os.version	No	Specifies the version of the targeted operating system.
variant	No	The variant of the specified CPU architecture. Example: ARM 32-bit, v7
config	No	Specifies runtime and execution options which apply when the image is run as a container. Includes nested objects.
config - User	No	The username or UID which is a platform-specific structure that allows specific control over which user the process runs as.
config - ExposedPorts	No	A set of ports to expose from a container running this image.
config - Env	No	A key-value array of strings in VARNAME=VARVALUE format that will become environment variables in the spawned container.

Property	Mandatory	Description
config - Entrypoint	No	A list of arguments to use as the command to execute when the container starts.
config - Cmd	No	Default arguments to the entry point of the container.
config - Volumes	No	A set of directories describing where the process is likely to write data specific to a container instance.
config - WorkingDir	No	Sets the current working directory of the entry point process in the container.
config - Labels	No	The field contains arbitrary metadata for the container.
config - StopSignal	No	The field contains the system call signal that will be sent to the container to exit.
rootfs	Yes	The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the file system hash.
rootfs - type	Yes	Must be set to <i>layers</i> , which is the only accepted value.
rootfs - diff_ids	Yes	An array of layer content hashes <i>DiffIDs</i> , in order from first to last.
history	No	Describes the history of each layer. The array is ordered from first to last.
history - created	No	Creation timestamp of the layer.
history - author	No	The author of the build point.
history - created_by	No	The command which created the layer.
history - comment	No	A custom message set when creating the layer.
history - empty_layer	No	This field is used to mark if the history item created a file system diff. Can be set to <code>TRUE</code> or <code>false</code>

Table 3.1.: Properties of an image configuration as specified by the *OCI*, adopted from (OCI, 2016d)

A JSON-structured example of such a configuration can be found in Appendix A.2.1.

The third main component of an *OCI* image are the file system layers. As already stated before, the file system of an image consists of different layers which stack up on each other, creating the file system that will be used within the execution environment of a container using the image. Each layer consists of compressed file types (regular files, directories, sockets etc.) which must support a subset of file attributes. The initial layer always defines the base for the other layers and can be seen as the starting point. It can be an empty directory, for example, created as `rootfs`. Other files and directories may now be created within the empty directory. Once the changes are completed, a snapshot of the layer will be taken in the form of a plain `tar`-archive. To create a second layer upon it, the first layer has to be expanded into a new directory preserving the file attributes, so that the new directory is an exact copy of the first one. Now one can perform changes in that new directory. Possible changes defined by the *OCI* are added, modified and deleted. It is important to represent these changes as only the changed files will be part of the layer that will be created out of the new directory. The new layer is defined as *changeset*, meaning that the files in the archive will be applied to the base layer rather than extracted. To specifically mark removed files, the full qualified file path will be prefixed with `.wh.` which is an abbreviation for *whiteout*. The layers are marked as distributable layers by default, meaning that they might be uploaded and downloaded without legal restrictions. If such a legal restriction applies to a certain layer, the layer may be flagged as non-distributable using the corresponding media type. Using that flag, a layer might be downloaded by a specific distributor but never be uploaded (OCI, 2016b).

Finally, the *image-spec* describes the optional component *image index*. An index serves as a pointer to image manifests and is mostly used when an image supports multiple architectures (OCI, 2016f). An example image index can be found in Listing A.2 which contains two references to image manifests that declare different architectures.

3.2.3. containerd

As there are many *container runtimes* existing which are worth mentioning, this section will focus on *containerd*, as it is an important part of the analysis taken in the further work. Graduated by the *CNCF* and therefore a member of the project, *containerd* is an API-driven *container runtime* that works as a daemon on *Linux* and *Windows* systems. It manages the life cycle of containers by operating the transfer and storage of images, execution and supervision of containers, the network settings and many more features that are necessary to run containers on a host. The units that cover the operational tasks

are separated into components and subsystems. To actually run containers on a host, a low-level *container runtime* needs to be used. All *container runtimes* that comply with the *OCI* specification may be used, but the default one is *runc* (OCI, 2015c). Per design, *containerd* is meant to run as a daemon and therefore not to be interacted with directly through a command line tool but rather via APIs. To achieve this, *containerd* exposes the remote procedure call framework *gRPC* (CNCF, 2014) as an API via unix sockets (containerd, 2015a).

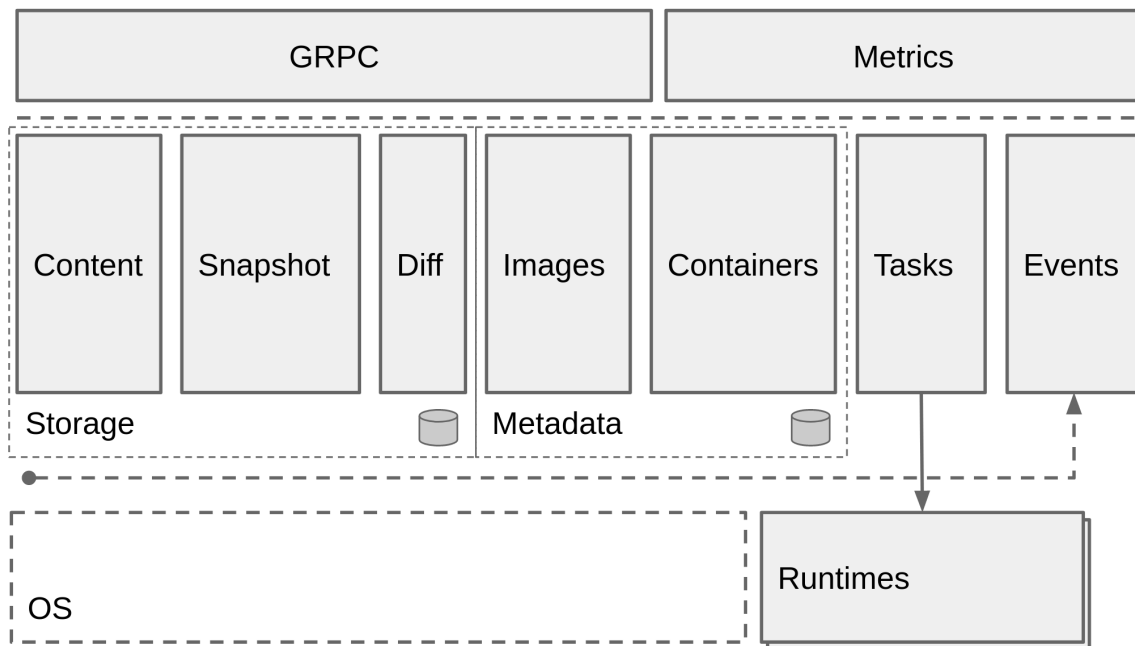


Figure 3.3.: Architecture of *containerd*, taken from (containerd, 2015b)

A container image as specified by *OCI* is defined as a collection of resources in *containerd* which is used to create a bundle. The bundle consists of the expanded file layers declared in the container image and a configuration file that contains the runtime information for the container. The actual configuration file of the image will be transformed to a target format used by *containerd*. To streamline the process of pulling an image, local store components are used. When an image gets pulled, the first operation is to fetch the manifest of the image, verify and store it within the *metadata store*.

Each layer stated in the manifest will now be fetched by its digest, verified and stored within the content component. As a second step, the *bundle controller* fetches the previously gathered information and creates a *snapshot* which will represent the *bundle's* `rootfs` file system and may as well include mount points.

To start and run a container, a *shim* is used as parent process of the container to intercept

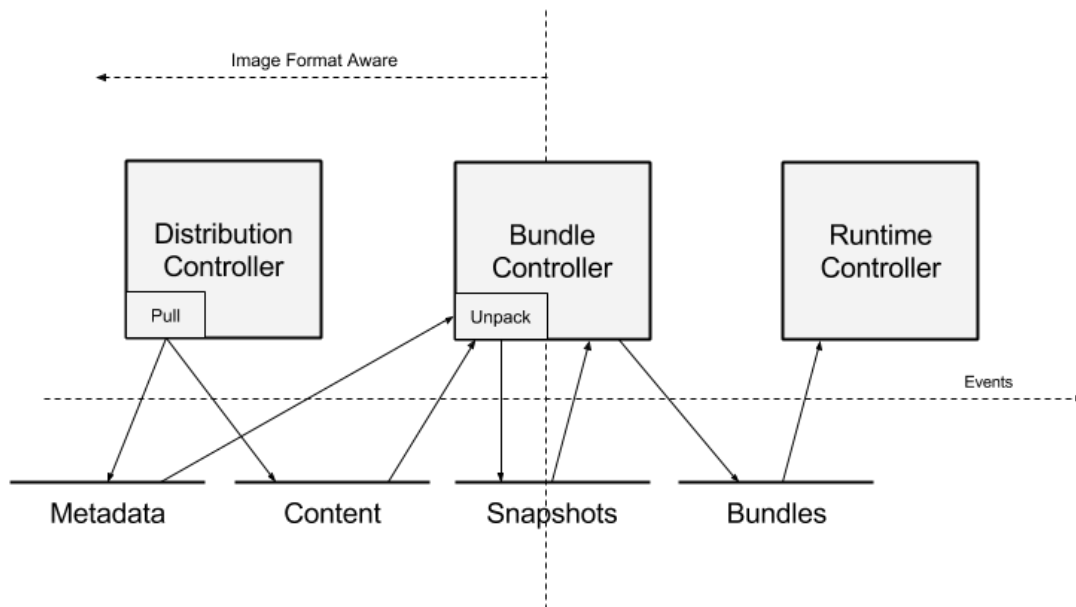


Figure 3.4.: Data flow of an image pull operation in *containerd*, taken from (containerd, 2015f)

input and output operations as well as fetching the exit status of a container so that it can be written to the *containerd* daemon. That once again clarifies that the daemon does not directly own the containers but rather manages their life cycle through APIs (containerd, 2015d). The *Task* component, as seen in Fig. 3.3, is used to create a running process on the host system itself while being managed by the *container runtime* (containerd, 2015g, see "Creating a running Task"). Usually it takes a container object and starts a new process on the system itself which actually runs the *container* (containerd, 2015a).

In order to finally use *containerd* as a *container runtime* for *Kubernetes*, the *container runtime interface (CRI)* (Kubernetes, 2016) of *Kubernetes* has been adopted by *containerd*. Each node in a *Kubernetes* cluster contains a component called *kubelet* which manages the life cycle of a *Pod* (see Section 3.3.1). To communicate with the underlying system of that node, the *kubelet* acts as a *gRPC* client and needs its *CRI* API to be implemented. This API is implemented as a plugin into *containerd* and therefore enables it to serve as valid *container runtime* on *Kubernetes* cluster nodes (containerd, 2015c).

3.3. Kubernetes

One famous tool that leverages containerization to deploy huge workloads at scale is *Kubernetes*. Originally developed by *Google* it has been donated as a project to the *Cloud native computing foundation* in 2014, thus being an open source software since then. Generally speaking, *Kubernetes* is an orchestration tool that allows to manage container deployments including the configuration of the infrastructure needed for the components to run. Instead of setting up the environment for a single container and deploying it to a hosting machine via usage of command line interfaces, *Kubernetes* offers to use declarative languages to define the needed infrastructure and components. This being said, an engineer does not necessarily need to know the details of a *container runtime*, but rather focus on the description of a desired infrastructure while *Kubernetes* is processing the definition. To provide such infrastructures, *Kubernetes* covers several functionalities, such as service discovery and load balancing. Ports defined on containers can be exposed using a DNS name or the IP address of a container. Given that multiple instances of a container can be deployed to cover high-availability scenarios, a load balancer handles incoming network traffic and distributes the communication across the sets. Furthermore, a controlled storage orchestration is provided. Local storage on the physical machine can get leveraged as well as storage provided through cloud services. Having the desired state defined also allows to roll out changes to the ecosystem. An upgrade of a containerized application can be done by simply changing the desired version and *Kubernetes* will deploy the upgrade while keeping other resources like storage or IP address assignments. A rollback does work in the opposite way. In case a deployed container enters an unhealthy state, *Kubernetes* can be configured to just remove and redeploy it, providing a self-healing functionality. In addition to that, a management of configuration and secrets is provided. Configuration sets that do not necessarily belong to *Kubernetes* but rather to the deployed application itself can be stored in so-called *ConfigMaps* which store key value pairs of information that will be injected to a container as environment variables. Classic configuration files can be stored in managed storage that gets mounted into the containers. Application secrets are also an important part of distributed deployments. Secrets can be stored in a secure way so that they do not get exposed in the configuration. Even though it contains a huge set of functionalities to maintain huge environments of multiple containers and services, *Kubernetes* itself needs an operating system to run on as it can be seen as an application when it comes to classic computing. This being said, the deployment of *Kubernetes* itself might contain one single or multiple nodes that have to

be managed and secured (Kubernetes, 2021r).

3.3.1. Architecture and Components

The composed set of various *Kubernetes* components forms a cluster. To get the cluster installed, it needs to get distributed across host machines called nodes. Based on their purpose those nodes will host the required *Kubernetes* components. In a distributed environment a master node will, for example, host the *Control Plane*, whereas the worker nodes will host the *kubelet* which runs the workload on the nodes. A single node distribution of a cluster is also possible, in which all *Kubernetes* components are hosted on a single node.

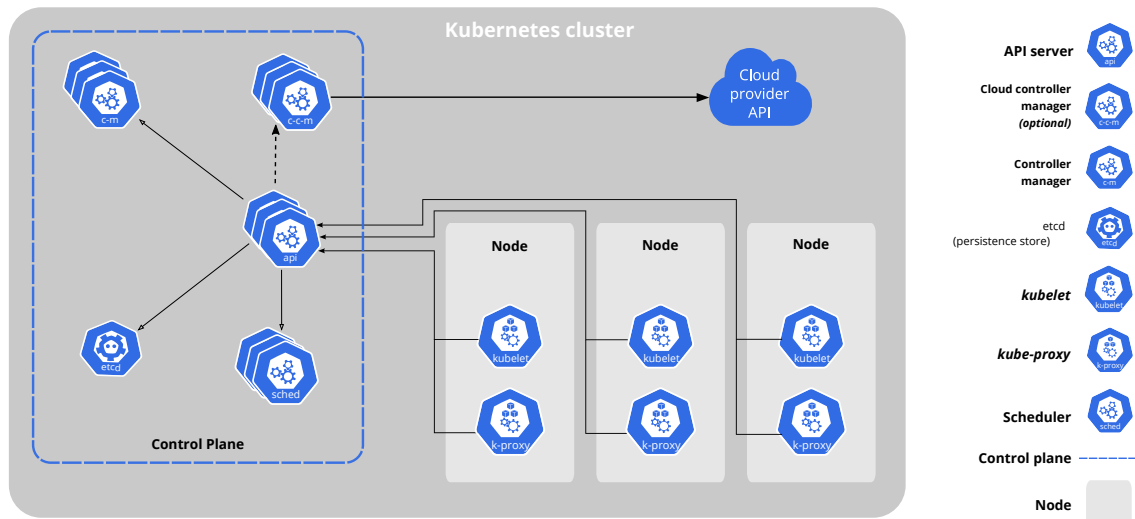


Figure 3.5.: The components of a *Kubernetes* cluster. Taken from (Kubernetes, 2021r)

As shown in Fig. 3.5, the *Control Plane* contains the most components. The several components could also be distributed via different nodes, so that the *Control Plane* itself is deployed in a distributed way, but are usually kept on a single node (forming the master node) while assigning user containers or other workloads to the worker nodes through *kubelets*. Furthermore, it collects information from within the whole cluster and invokes commands needed to satisfy the desired state. Being a standalone component, the *kube-apiserver* can get deployed in multiple instances to achieve high availability and balance load when the cluster's size increases. The place where *Kubernetes* stores all its cluster data is called *etcd*, which is an individual project itself (etcd, 2013b) maintained by the *CNCF*. The information in *etcd* is stored in key-value-pairs and again, this standalone component can be deployed with multiple instances in a distributed environment.

The scheduling of workloads such as assigning *Pods* to nodes is handled by the *kube-scheduler*. Other factors like memory or CPU requests, (node-)affinities etc., are also taken into account for the process of scheduling. The monitoring of the shared state of a cluster and comparing it to the desired one lies within the responsibility of the *kube-controller-manager* component, which operates a single controller process that monitors different types of objects like nodes, jobs or service accounts. When a configuration drift has been detected, a change will be requested via the *kube-apiserver* and processed by the affected components. Besides the *kube-controller-manager*, there is the *cloud-controller-manager* which is used for cloud deployment scenarios. It contains specific API implementations that are tailored to the cloud vendor and separate the communication of components that interact with cloud services from components that interact with the local cluster only. If there are no cloud services used, which would apply to on-premise deployments, the *cloud-controller-manager* is not part of the cluster.

Besides the components that are mainly part of the *Control Plane* and therefore reside on the main node, there are two components that are part of worker nodes as well. The *kubelet* component is responsible for running the containers which are part of a *Pod*. Each node in a *Kubernetes* cluster contains a *kubelet*. It takes the specification of a *Pod* and processes the desired state, creating the desired containers and operating them by using the underlying *CRI* implementation. Therefore, each node needs to have a *container runtime* installed which is then being instructed to run the container as a process on the host (see Section 3.2). Another component that runs on each node is the *kube-proxy*. Its purpose is to take care of the communication between the node on which it is installed and the cluster. Therefore, it leverages the packet-filtering capabilities of the underlying operating system, but could also forward traffic by itself if none are available (Kubernetes, 2021h).

3.3.2. Kubernetes Objects

While there is a various subset of objects that can be deployed and configured in *Kubernetes*, this subsection focuses on those that are used or seen within the work of this thesis. The most common objects in a cluster are *Pods* (Kubernetes, 2021m). This object defines as the smallest deployable unit in *Kubernetes* and includes a set of containers that will share the same resources of the operating system, namely the *Linux Namespaces* (see Section 3.1). A common deployment method is to run one single container per *Pod*, wrapping an application. This enables *Kubernetes* to operate the *Pod* rather than the sin-

gle containers. There are also scenarios where it makes sense to deploy multiple different containers in the same *Pod* which work tightly together. In an example scenario, one container might host files from a shared volume to a public instance while another container, often referred to as *sidecar* container, works with the files and has more permissions than the hosting container in order to perform write operations. As *Pods* sometimes might need more objects to interact with, they get specified within an object called deployment. This resource contains the desired state of multiple objects that need to be deployed within a cluster forming an overall application set. In order to deploy the same *Pod* multiple times to enable load balancing scenarios, a *ReplicaSet* (Kubernetes, 2021n) can be used which tells the *kube-controller-manager* to deploy more instances of that resource. As stated before, a *Pod* uses resources provided by the *Linux* namespaces which allows to also access the hosts file system via *storage volumes*. Given that there are multiple types and complexities of *storage volumes*, it is important to differentiate between two kinds: *PersistentVolume* (Kubernetes, 2021l) and *volume*. The first one allows for *storage volumes* to be persistent. This needs to be requested via a *PersistentVolumeClaim* and will be assigned to a worker node directly, therefore creating a node affinity. Even if all resources that use such a persistent *storage volume* get deleted, the *storage volume* itself will not get deleted. All contents will be kept so that newly created resources can access them again. This scenario is often used in web application environments, where it allows to recreate *Pods* while keeping the stored data and mounting them back into the containers. When not explicitly defined as *PersistentVolume*, the resource will be deleted as well when, for example, a whole deployment gets recreated. If not specified otherwise, a container will not run with privileged permissions. One can specify within the *Pod* deployment to run a container as privileged, which will add the `privileged` *Linux*-flag. This allows the container for more capabilities, such as accessing hardware devices. This can lead to security issues. In a scenario where a *storage volume* is located in the worker node and mounted into a privileged container, an attacker could misuse that privilege and try to access file system paths that should normally not be visible to that container (Kubernetes, 2021q).

Besides storage, *Pods* also share the network resource with the underlying node. Each *Pod* gets its own network namespace containing unique addresses. This means that the container(s) deployed in that *Pod* share the same IP address and network port(s) as well. They communicate with each other using `localhost` or inter-process communication mechanisms provided by the operating systems. If the containers need to reach other *Pods* or the public Internet, this needs to be configured via cluster networking resources

(Kubernetes, 2021d). One of these cluster networking resources is the *Service*, which tackles the definition of network communication to and from a *Pod*. As each *Pod* gets a unique IP address, it would be hard to maintain the communication between applications when recreating a deployment. A *Service* is a simple resource definition that contains assignments and access policies to *Pod* resources. In a scenario where an application is deployed with load balancing, thus using *ReplicaSets*, each *Pod* instance has an own IP address while the application is exposing the same port. Now, a selector can be assigned to the *Pod* description which is used by the *Service* to find that service and expose it. In this example scenario an application could expose the port 1234 and the *Service* is configured to map this port to a cluster-wide port 80. All of the *Pods* specified within the *ReplicaSet* (all instances) are then reachable through port 80 and the *Service* knows how and where to route that traffic, to make sure to always reach a *Pod* hosting the application (Kubernetes, 2021p).

Another important resource are *Secrets* (Kubernetes, 2021o). They provide a way to store confidential information in a dedicated place and not directly within an application or container image. *Secrets* are used by *Pods* and are implemented by different methods. They can be used as files within a *storage volume* and then get mounted into the container(s). The specified name of a *Secret* will become the file's name and the content would be the actual confidential information. A container that needs to consume that information has to read the file's content. Another way to use *Secrets* is to mount them as environment variables into a container. They can then be accessed easily via the *Secrets*'s name, which also defines the name of the created environment variable. The third way is to use *imagePullSecrets* but, other than the two methods mentioned before, this method applies to secrets that are used to pull images from (private) container registries only. The *Secrets* will be handed over to the *kubelet* which will then become able to pull images on behalf of a *Pod* (ibid.).

The *ConfigMap* resource is similar to the *Secret* resource. A main difference is that *ConfigMaps* are not intended to store sensitive data but rather configuration data for services and applications included in a *Pod*. Therefore, they also provide additional data types, i.e. `binary data`, which may store base64-encoded binary data and `data` which can store UTF-8 byte sequences. The ways to deploy *ConfigMaps* to *Pods* are the same as for *Secrets*. As the *ConfigMaps* have a size limit, it is recommended to store large configuration items or files via *storage volumes* and not via a *ConfigMap* (Kubernetes, 2021e).

3.4. Used Tools

3.4.1. Virtualbox

VirtualBox Oracle, 2021 is an open source virtualization software that runs as a *hypervisor* on a vast amount of operating systems. It provides functionalities to abstract and virtualize basic system hardware components like the CPU, RAM and storage components so that it forms an isolated environment for a virtual machine. A guest operating system can be installed on that virtual machine without affecting the host system components. This approach is used within the work of this thesis to host a *Kubernetes* cluster on virtual machines. *VirtualBox* allows to automate several tasks and operations through the `VBBoxManage` binary. This is used within the implementation of the general analysis process to automate all necessary steps that contain interaction with *VirtualBox* as explained in Chapter 4. The used *Kubernetes* cluster will contain two virtual machines as member: One virtual machine gets the role of a master node that hosts the *Control Plane* and the other virtual machine will be declared as worker node, which hosts the necessary components to run a container (see Section 3.3.1). The term *master* is used to describe the machine that manages the whole communication in the cluster. The *worker* takes an assigned workload and creates the specified resources. To automate the creation and configuration of the master and worker node, *Vagrant* is used together with *Ansible*.

3.4.2. Vagrant

Vagrant (HashiCorp, 2010a) is a tool developed and maintained by *HashiCorp* that automates the creation of virtual machine environments. The source code is publicly available, thus being an open source software (HashiCorp, 2010b). It provides the functionality of storing the configuration of virtual machines in a so called *Vagrantfile*, following the approach of *Configuration as code* or *Infrastructure as Code*. Many virtualization software vendors are supported to be used with *Vagrant*, such of one is *VirtualBox*. The configuration of the virtual machines that form the *Kubernetes* cluster used in this thesis is stored as code. Using this approach, the environment is documented in a human readable format and can easily be adopted on other host systems that have *Vagrant* and *VirtualBox* installed. *Vagrant* also provides enhancements to provision virtual machines after they have been created. This includes the installation of software and other necessary steps to bring the machines into a declared state. One of those integration's is *Ansible* (see

Section 3.4.3). The used *Vagrantfile* in this thesis can be found in Appendix A.1.3.

3.4.3. Ansible

Ansible is an open source automation tool sponsored by *RedHat* (RedHat, 2012). It can be used to automate the configuration of various software or infrastructure components, from a basic operation system to large scaled cloud environments. The tool is developed in *Python* language and works without the requirement to have an agent installed in order to manage a component. Instead, an SSH connection is used and wrapper functions will run specific commands on the managed machine to perform operations. In the further course of the work, *Ansible* is used as a provisioner-plugin of *Vagrant* to automate the configuration of a *Kubernetes* cluster. The configuration is stored in *roles* that apply to the virtual machines, where the *role* for master and worker node do differ. This implementation gets also described in Section 4.2.1. The used files that contain the *Ansible roles* can be found in Appendix A.1.3.

3.4.4. Kubectl

The command line tool *kubectl* (Kubernetes, 2021g) provides the functionality to control and interact with a *Kubernetes* cluster. It is a wrapper for the API function calls and is used to perform various operations on a cluster, like create, list or remove *Kubernetes* objects and other resources from commandline. A *kubeconfig* file is needed in order to be able to communicate with a *Kubernetes* cluster, which contains a necessary token for authentication on other information like the IP address of the *Control Plane*. Within this work the *kubectl* tool is used to deploy, configure and remove resources from the cluster and integrated in the automated analysis process. It needs to be installed on the host machine that runs the analysis and will therefore communicate with the cluster that is distributed across the virtual machines. An installation is supported for *Windows*, *Linux* and *MacOS*. The analysis framework will check automatically if a *kubeconfig* file is available. If not specified as an environmental variable, it will be downloaded from the master node before the analysis starts.

3.4.5. PowerShell

PowerShell is a cross platform scripting language based on the *.NET Common Language Runtime* developed by *Microsoft*. It ships with an inbuilt shell and is mostly used for task automation. Originating from the *PowerShell* dedicated to *Windows* operating system, a core version of *PowerShell* has been released as open source software in 2016 by *Microsoft* and became available to other operating systems too. It follows the *Verb-Noun* principle for functions, which are also referred to as *cmdlets*. For the sake of readability the term *cmdlet* will be named *commandlet* in this thesis. It also allows to collect *commandlets* in modules that can be imported to have them available during runtime. *PowerShell* will be used in this work to automate the analysis process as described in Section 4.2.2. This enables the integration of the analysis process on all operating systems that support *PowerShell* in its core version. To automate the analysis process, two modules have been developed for this work: *Vbox4Pwsh* and *PowerForensicator*, which will be explained in detail in the aforementioned section.

3.4.6. Autopsy

Autopsy is an open source digital forensics platform originally developed by Brian Carrier in 2008 (Carrier, 2021), that provides a graphical user interface to the *sleuthkit* binary tools. It allows to work with *cases* where data sources such as raw disk images, a loose set of files and other digital evidences can be added to. The evidence will be imported, parsed and can be analyzed by leveraging a subset of integrated or third-party plugins. Within this work *Autopsy* is used to analyze exported raw disk images from the *VirtualBox* machines in order to perform a post-mortem analysis. The term *post-mortem* originates from the context of digital forensic investigations and is used within this work to describe the state after the execution and completion of a specific action. Mainly the keyword search and file system analysis functionalities of *Autopsy* are used within the result sections in Chapter 5.

CONCEPT

4.1. Analytic approach

As described in Chapter 2, the analytic steps taken will be based upon the approach of differential forensic analysis as worked out by Garfinkel et al. (2012). Following this approach, the different states of an object will be compared to each other in order to find out what changed on the way from state A to state B. An object can be anything from a single file to a raw disk image. Primarily, disk images will be analyzed within this work. It is necessary to create a baseline image that defines the initial state or starting point for an action, and a final image that defines the state of a processed action. Changes within the file system of an image will be detected by analyzing the timestamps and content changes of the files. To do so, the *DFXML* toolset will be used. It provides the functionality of parsing a raw disk image with *fiwalk* from the *sleuthkit* tools (Carrier, 2011) and comparing two images to each other. The detected differences will be brought into a machine-readable XML format so that the results can be used further (Garfinkel, 2012b). The main objective is to find characteristic evidences as defined by Dewald (2012) within the file system that only occur when a specific action has been taken. Those results will define the action evidences. Therefore, an additional image will be created which contains the background noise of the system where the analysis takes place. The base state will be the same as for the action but, other than for the action state, no manually crafted interactions will happen. The system will start at the base image's point and run in idle state for a specific time frame. The differences between the base image and the final image will be analyzed in the same way as for the action state and define the noise evidences. In order to get the characteristic evidences, the noise evidences will be subtracted from the

action evidences. The final result will contain only those differences that can be assigned to the executed action, containing the file path within the image and the timestamps that have changed. Given that the images have been exported, the files can then be extracted from the action image so that a deeper analysis can be performed, like reviewing the file's contents etc. In case a file, for example, got modified, the initial file can be exported from the base image as well. In addition to the file system analysis, the file system mount points of a virtual machine will be parsed. Especially for *Kubernetes* nodes, it is important to know if mount points are affected by specific actions. The analysis process of mount points is similar to the one of file systems and will be explained in detail within the next sections.

4.2. Automation

The major requirement for the analysis is to be repeatable. Therefore, the setup of the virtual machines and the configuration of the cluster need to be stored as code so that one can reuse the codebase to recreate the setup used within this analysis. Furthermore, the whole process of the differential analysis needs to be automated so that it can be repeated several times. This will be done using *PowerShell* as a scripting language.

4.2.1. Cluster Provisioning

In order to maintain full control over the involved systems, the *Kubernetes* cluster will be provisioned on virtual machines using *VirtualBox* (see 3.4.1) as virtualization software, as it allows to perform several actions like the export of a disk image in a state of a specific snapshot or dumping the system's memory.

When it comes to *Kubernetes*, a distributed deployment model is often used, where the master and the worker node are hosted on separate instances. In order to track changes that are specific for the master or the worker node, this model will be used for the following analysis. Using *Vagrant*, two virtual machines will get created in *VirtualBox* - one for the master node and one for the worker node. The configuration of the cluster itself is done by an *Ansible* playbook which first configures the cluster on the master node and then joins the worker node into that cluster. The underlying container runtime used is *containerd* (see Section 3.2.3). When the creation and configuration is done, the master node contains a `config` file which will be used to access the cluster with `kubectl`

from the host. Given that `kubectl` can then be used to interact with the cluster, possible automation scenarios can and will be applied.

4.2.2. Analysis with *PowerShell*

For the purpose of automation, two *PowerShell* modules were developed: *Vbox4Pwsh* (see Appendix A.1.1) and *PowerForensicator* (see Appendix A.1.2). Both modules can be used with the open source core version of *PowerShell* and will therefore work on both *Linux* and *Windows*.

Vbox4Pwsh is an abbreviation for *VirtualBox for PowerShell* and contains commandlets that allow to interact with the locally installed *VirtualBox* software. The module detects whether it runs on *Linux* or *Windows* and behaves accordingly - given that *PowerShell* is installed on the system. The module is basically a wrapper for the `VBoxManage` binary that allows to perform a variety of actions on and with virtual machines. In order to use those functionalities within the analyzing scripts, the following commandlets have been developed:

Name	Purpose
Copy-VBoxDiskMedium	Clones a VBox disk medium from source to target.
Close-VBoxDiskMedium	Closes a disk medium in VBox.
Copy-FileFromVBoxVM	Copies a file from a VBoxVM to the local machine.
Copy-FileToVBoxVM	Copies a file from local source to VBoxVM.
Copy-VBoxDiskMedium	Clones a VBox disk medium from source to target.
Get-VboxHDDs	Retrieves all media declared as HDD from VirtualBox.
Get-VBoxVMInformation	Retrieves information about a VM and returns a readable list.
Get-VboxVMs	List all VMs configured in Virtualbox.
Get-VBoxVMSnapshot	Gets the snapshots of a VBox VM.
Invoke-Process	This is a little helper for creating a 'good' process. Start-Process is not sufficient here.
Invoke-VboxVMProcess	Runs a program on the VM.
New-VBoxVMSnapshot	Creates a snapshot of a VBox VM.

Name	Purpose
Remove-VBoxVMSnapshot	Removes a snapshot from a VBox VM.
Restore-VBoxVMSnapshot	Restores a snapshot of a VBox VM.
Resume-VBoxVM	Starts a VBox VM that is in a saved state.
Save-VBoxVMState	Saves the state of a running VM.
Start-VBoxVM	Starts a VBox VM if it is in a stopped state.

Table 4.1.: Commandlet overview of the *Vbox4Pwsh* module

Once the module has been imported, the commands can be used within *PowerShell*. While the commandlets could also serve generic use cases for administration purposes, they are mainly used within the second module *PowerForensicator* for this work, which automates the phases of the analysis process.

The commandlets of that module take care of the steps which are necessary to generate the evidences and respectively filter for characteristic evidences. They are used within a single script file that contains all steps that are necessary for the analysis of a use case. Detailed functionalities of the single commandlets as well as the technical procedure of the analysis process will be described within the next section.

The files needed for a use case analysis will be stored separated from that module, in order to keep a structured overview and distinct between the modules and the actual use case scenarios. This allows to have a distinct script file containing the analysis steps for each scenario.

4.3. Analysis process

The objection of this section is to show how the procedure of analysis has been technically implemented and how the characteristic evidences will be filtered. In general, the process is divided into three different phases: *Init*-, *Noise*- and *Action*-phase. Within the *Init*-phase, the necessary actions are taken which will bring the virtual machines into a state from which actions will be performed. To achieve this, one can add customized actions to the analyzing script of that particular scenario. In order to catch the cached file system changes as well, the commandlet `Invoke-SyncOnVM` will run the binary `sync` on a virtual machine in order to write all cached changes to the disk. When the

actions have been taken, the state of the virtual machines will get saved and a snapshot called *Init* will be taken that can be restored whenever the machines need to be brought back into the initial state. To save the state of the machines and take a snapshot, the commandlets `Save-VBoxVMState` and `New-VBoxVMSnapshot` are used. Furthermore, the disk state of that snapshot will be exported as a bit-wise cloned image in raw format, using the exporting functionalities of *VirtualBox* provided by the commandlet `Clone-VBoxDiskMedium`. Optionally, the file system mounts of the virtual machine can be obtained by the commandlet `Get-FSMounts`. This will run the binary `findmnt` on the virtual machine and parse the output into a *PowerShell* custom object. This information will be stored into a variable that is used as reference for the upcoming phases.

The second phase *Noise* aims to collect the background noise of the virtual machines. The *Init* state will be restored and the machines will run in idle mode for a specific set of time, e.g. 5 minutes. Restoring a snapshot is done by the commandlet `Restore-VBoxVMSnapshot` which takes the name of a snapshot as input parameter. Directly followed by that, the commandlet `Start-VboxVM` will resume the virtual machine from the saved state. When the set time has elapsed, the state of the machines will again get saved and a snapshot called *Noise* will be taken. Before that, the file system mounts can again get obtained. A difference between the mount points of *Init* and *Noise* can be analyzed through the commandlet `Compare-FSMounts`, which takes the variable generated within the *Init* phase as a reference and the newly generated output of *Noise* as input and resolves the differences. New mount points will be marked with the term *ADDED* and removed mount points will get *REMOVED*. If a mount point exists within the reference object as well as within the difference object, the properties `Source`, `FSType` and `Options` will be looked at so that changes within the properties are detected. Again, a raw disk image containing the snapshot's state of the virtual machines will be exported. As there are now two raw images per machine, one containing the *Init* state and one containing the *Noise* state, the differences will be analyzed. The extraction process as well as the differential analysis is done through the commandlet `Invoke-ActionStateFileExport`, which saves the virtual machines state, creates the snapshot, exports the raw file and then invokes the tools `fiwalk` and `idifference2.py`. Based on the parameters given to that function, the tools can either be invoked locally on the host system or get invoked within an isolated docker container, so that the host system does not necessarily have to have the tools installed, as long as the docker container can be started. If `fiwalk` gets invoked locally instead of through `idifference2.py`, the resulting XML file will be stored and parsed as

input file to `idifference2.py` instead of the raw images. Regardless of that, an XML file of the *Init* raw image will be created in any case. This brings in the advantage of only parsing the *Init* raw image once it got created, instead of every time that `idifference2.py` gets invoked as the XML file is much smaller than the raw image file. Furthermore, it enhances the robustness of the overall process because the *Init* XML file won't get changed, whereas when both *Init* and *Noise* raw files are given as input to `idifference2.py`, `fiwalk` will run over the *Init* raw file again for each invocation. The result of the `idifference2.py` script will be saved to a dynamically named file called `/<BaseDir>/NOISE/<VMName>_Noise.idiff`. To already reduce the duration of the raw image export and generation of `idiff`-files, the commandlet parallelizes this procedure for each virtual machine included in the analysis.

Once the `idiff`-file has been generated, the XML-formatted content will get parsed into a simpler format which only contains the file object's path and the modification's records reported by `idifference2.py`. The reported modifications will be translated as follows:

Record	Explanation
a	The <code>atime</code> (accessed) timestamp of the file did change.
c	The <code>ctime</code> (changed) timestamp of the file did change.
m	The <code>mtime</code> (modified) timestamp of the file did change.
cr	Describes that a file got created.
d	Describes that a file got deleted.
r	Describes that a file got renamed.

Table 4.2.: Meaning of the reported file records

An example could be a file that got created during the *Noise*-phase, for which the text will look like this: `/path/to/file cr`. The parsing of that `idiff` file is done by the function `Format-ActionEvidence`, which will save a resulting text file named `/<BaseDir>/NOISE/<VMName>_Noise-Evidence.txt`.

Given that the *Noise*-phase has been successfully concluded, the last phase called *Action* will start. The steps performed within this phase are similar to those within the *Noise* phase.

The only difference is that, instead of letting a virtual machine run in idle state for a specific amount of time, the actual action that is to be analyzed will be performed.

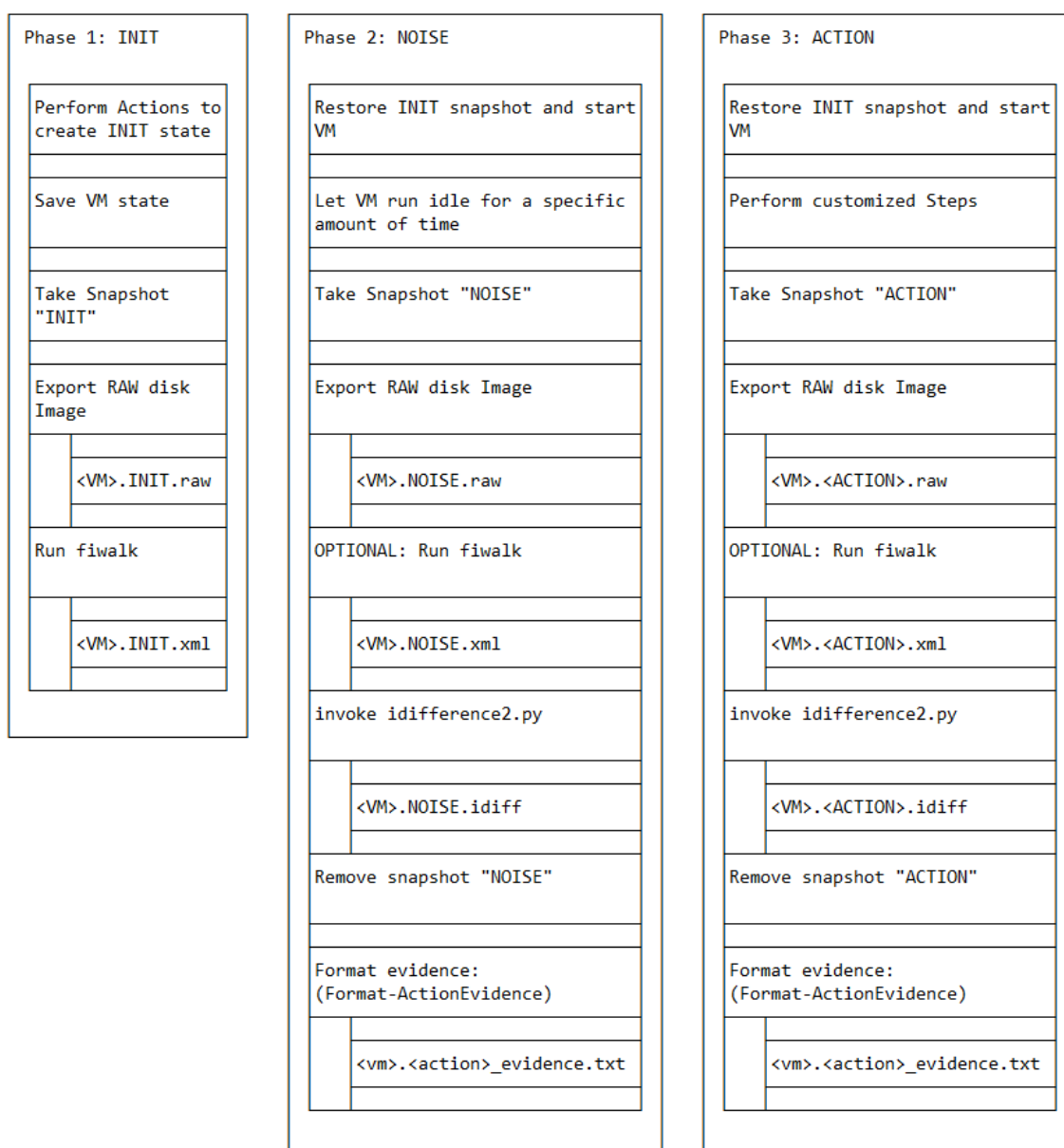


Figure 4.1.: Overview of the analysis process

When both phases are finished, the function `Get-CharacteristicEvidences` is used to parse the created evidence files. The contents of *Noise* will get subtracted from the *Action* evidence's contents. Both the *Noise*- and *Action*-phase can be repeated several times. Running these phases multiple times is crucial to the overall filter for characteristic evidences. If the phases have been repeated 5 times, for example, only those entries within the evidence files that occur exactly 5 times will be taken. The commandlet also takes an optional parameter called `MinCount` so that the minimum count of an evidence can be lowered. In an example where the parameter is set to four but five rounds were taken,

also those evidences with a minimum count of four would be seen as characteristic. In case the file system mounts were extracted as well, the characteristic file mounts will also be calculated following the same approach.

4.4. Example

To demonstrate how the analysis process can be implemented and to verify the results, an example scenario will be created. The use case displayed by this scenario is to observe file modifications of a certain set of files that will be manually created for the *Init* state and will get changed during the *Action* phase. Based on the file modifications, a certain result can be expected. Even though the performed file operations are not related to *Kubernetes* directly, they will be invoked on both the master- and worker-node.

To create the *Init* state, a shell script is copied to both virtual machines which will create files that serve a specific purpose within the later *Action*-phase. The purpose of each file is written to its content.

```
1  #!/bin/bash
2  # Purpose: Create a few files that shall exist for the INIT snapshot
3  mkdir ~/Documents
4  echo "This is a test file. Delete the file!" > ~/Documents/delete.me
5  echo "This files atime will get changed by 'touch -a!'" >
   ↳ ~/Documents/access.me
6  echo "This files mtime will get changed by 'touch -m!'" >
   ↳ ~/Documents/modify.me
7  echo "This files atime AND mtime will get changed by 'touch'" >
   ↳ ~/Documents/touch.me
8  echo "This files ctime will get changed through 'chmod +x'" >
   ↳ ~/Documents/chmod.me
9  echo "This is a test file. Change the content!" >
   ↳ ~/Documents/change.me
10 echo "This is a test file. Append content!" > ~/Documents/append.me
11 echo "This file shall get renamed!" > ~/Documents/rename.me
12 sleep 2
```

Listing 4.1.: Files created for *Init* state

Once the files have been created according to the commands in Listing 4.1, the *Init* snapshot will get taken, meaning that the files are already present within the *Init* state. The *Init* phase is followed by the *Noise* phase, which will just keep the virtual machines running in idle mode for a certain amount of time. When the phase has finished, the *Action* phase starts by modifying the created files. To do so, another script file containing `bash` commands will be copied to the virtual machines and executed.

```
1  #!/bin/bash
2  # Purpose: Change the files that have been created in the INIT
   ↳ snapshot
3  mv ~/Documents/rename.me ~/Documents/IhaveBeen.renamed
4  touch ~/Documents/create.me
5  rm -f ~/Documents/delete.me
6  touch -a ~/Documents/access.me
7  touch -m ~/Documents/modify.me
8  touch ~/Documents/touch.me
9  chmod +x ~/Documents/chmod.me
10 echo "This content got changed!" > ~/Documents/change.me
11 echo "This line has been appended to the file!" >>
   ↳ ~/Documents/append.me
12 sleep 2
```

Listing 4.2.: File modifications performed during the *Action* phase

For the specific purpose of changing only timestamps, the tool `touch` (Linux man-pages project, 2021m) is used. To only change the modification timestamp, the tool `chmod` (Linux man-pages project, 2021c) is used on a specific file. Having full control over the file operations allows to estimate characteristic evidences which will result out of this analysis. The expected evidences should look like the following:

File	Timestamps changed	Explanation
delete.me	d	The file gets deleted.
create.me	cr	The file gets created.
access.me	a	The <code>atime</code> of the file gets directly changed through <code>touch</code> .
modify.me	m	The <code>mtime</code> of the file gets directly changed through <code>touch</code> .
touch.me	mc(a)	Multiple timestamps will change for that file.
chmod.me	c	Only the permissions will get changed, not the contents.
change.me	mc(a)	The content of the file gets changed.
append.me	mc(a)	The content of the file gets changed.
rename.me	d(mc)	This file will be renamed, therefore the file itself might get deleted.
IhaveBeen.renamed	cr or r	As this is the renamed file <code>rename.me</code> it either gets the <code>created</code> timestamps or the <code>renamed</code> flag by <code>idiff</code> <code>ference2.py</code>

Table 4.3.: Expected characteristic evidences

The actions are performed 20 times, which means that an operation needs to deliver the same result for each round to be seen as characteristic.

The overall run time of this analysis was around 10 hours. The characteristic evidences are stored within the files `kubernetes-master-1-FileModifications-CharacteristicEvidence.txt` and `kubernetes-worker-1-FileModifications-CharacteristicEvidence.txt`, as both the main and worker node have been included within the analysis process.

The characteristic evidences processed for the file operations from both master and worker node are nearly identical with respect to the manually changed files (see Appendix A.3.1 and Appendix A.3.2). As an example, a deeper look into the lines 1-13 of Listing 4.3 will be taken:

```
1 home/vagrant mc 20
2 home/vagrant/00_PreInit.sh d 20
3 home/vagrant/02_FileMods.sh cr 20
4 home/vagrant/Documents mc 20
5 home/vagrant/Documents/access.me ca 20
6 home/vagrant/Documents/append.me mc 20
7 home/vagrant/Documents/change.me mc 20
8 home/vagrant/Documents/chmod.me c 20
9 home/vagrant/Documents/create.me cr 20
10 home/vagrant/Documents/delete.me dmc 20
11 home/vagrant/Documents/IhaveBeen.renamed rc 20
12 home/vagrant/Documents/modify.me mc 20
13 home/vagrant/Documents/touch.me mca 20
```

Listing 4.3.: Selected characteristic evidences of worker node

Most of the results shown in Listing 4.3 do reflect the expected results from Table 4.3. Other than expected for some files, the `ctime` changed for every single file. The result for `delete.me` does also contain `mc` which means that the `mtime` and `ctime` changed for that file. The entry in the `idiff` file as seen in Listing A.5 shows that the `<alloc>` value changed from 1 to 0, thus marking the file as removed. Furthermore, several other properties changed: `mtime`, `ctime`, `nlink`, `filesize` and both `hashdigest` entries. As the commandlet `Format-ActionEvidence` parses not all but some of those properties, it could be argued to ignore the timestamp properties when a `delta:deleted_file="1"` entry is given, which would resolve in `d` rather than `dmc` for that particular file. On the other hand, all changed timestamp properties shall be displayed in the characteristic evidences. Therefore, this method of not ignoring other changes on deleted file objects will be kept throughout this work. The file `IhaveBeen.renamed` is showing the changed properties `rc` where `r` stands for *renamed* and `c` for changed. Listing A.6 shows the `idiff` file entry for that particular file. No other properties changed. Again, it could be argued to ignore the change of `ctime` just like in the example before.

Comparing the expected results to the actual results proves that the technical implementation of the analysis is working and can be used for further analysis scenarios.

4.5. Difficulties

The most present difficulty is the run time of the overall analysis. The duration of exporting a raw file image out of *VirtualBox* highly depends on the hardware given on the host system. The higher the I/O rate of the physical disk, the faster the process of exporting a disk image. This applies to the process of running `fiwalk` and `idifference2.py` as well, as they both parse the exported raw disk image(s). Especially on *Windows* operating systems when using *Docker* to run those commands, the I/O rate depends on the implementation of *Docker* on *Windows*. If *Docker* is installed using the *WSL2* as a back-end, the performance of file system operations decreases when a *Windows* path is mounted into a *Linux* container via the *WSL2* (see Docker, 2021c, best practise). This could be mitigated by using both tools locally instead of a docker container, but at the time of writing it is not reliable enough to compile a `fiwalk` binary for *Windows*, even though the *PowerShell* modules would support running it locally. Those restrictions when using *Docker* would not apply on *Linux* systems as the implementation of volume sharing is done in another way.

Another caveat is the amount of storage that is needed. Especially if two virtual machines are to be analyzed in parallel, the necessary amount doubles. In an example where there are two virtual machines that both have a disk size of 120GB, the disk capacity needed to store the exported images would be 480GB - twice the *Init* image and twice the *Noise* or *Action* image to be parsed using `idifference2.py`.

Furthermore, it highly depends on the analyzed virtual machine how much background noise is generated within the *Noise* phase. If the analysis starts at a point in time where a basic update or another task is scheduled to run, that does not necessarily have something to do with the actual analysis. The evidences of that task might appear in the resulting characteristic evidences. This can be mitigated by reducing the background activity of the system to an absolute minimum, which can often be a hard task due to a lack of documentation.

ANALYSIS

5.1. Analysis Environment

As the goal of the analysis framework in *PowerShell* was to support both *Windows* and *Linux*, two host machines are used in parallel for the analysis. A physical host machine with *Windows* installed and a virtual machine with *Ubuntu* installed and hosted on *Azure*, the cloud environment of *Microsoft*, were used.

For the *Windows* machine it is important to install and activate the *Windows Subsystem for Linux (WSL)* in version 2. It enables the use of *Vagrant* and *Ansible* together in order to create and provision the virtual machines the actions will be performed on. When *Ubuntu* is used as a distribution installed within the *WSL*, one can invoke the tools from within the shell while operating with the virtual machines provided through *VirtualBox* on the *Windows* operating system (HashiCorp, 2010c).

An overview of the software components containing their versions can be found in Table A.6.

5.2. Scenario 1: Creation of a Pod

5.2.1. Description

This scenario takes a look into one of the most essential parts of *Kubernetes*: Deploying and running a container image. A *Pod* specification will be applied to a running cluster

that tells the *kubelet* to download the specified image manifest and to run it as a container. The image used for this purpose is the lightweight *busybox* image in its latest version. As of the time of writing, the tag `latest` is referring to the image `busybox:1.34.1` (library, 2015). It is maintained by the *docker-library* project which describes and maintains image files for commonly used programs. The specification is written in *YAML* notation and contains the name of the *Pod* to create, a name for the container within the *Pod* and the actual image identifier as well as a command that will be executed when the *Pod* has successfully been scheduled. The executed command will simply run `sleep 3600` which runs the container in idle mode for one hour.

```
1 # Taken and adopted from
   ↳ https://kubernetes.io/docs/concepts/workloads/pods/
2 apiVersion: v1
3 kind: Pod
4 metadata:
5   name: simple-pod
6 spec:
7   containers:
8   - name: simple-pod-container
9     image: busybox
10    command: ['sh', '-c', 'echo "This is my simple Pod" && sleep
   ↳ 3600']
11 restartPolicy: OnFailure
```

Listing 5.1.: Specification of the *Pod* resource, adopted from (ibid.)

The objective of this scenario is to find out what changes happen on the file systems and mount points of both the master and worker node. As the *Pod* should get deployed on the worker node, it is expected that the most changes will happen on the master node.

5.2.2. Analysis

All steps that are necessary to perform the installation are collected in a single `Analysis.ps1` file which reflects the three phases *Init*, *Noise* and *Action* as described in Fig. 4.1. Within *Init* and *Noise*, no manual actions are taken. The cluster just runs in idle mode until the *Action* phase is reached. Within that phase, the specification shown in 5.1 gets applied to the cluster by using the command `kubectl apply`.

```

218 $Pods = Invoke-Kubectl -kubectlArgs ("apply", "-f", [System.IO.Path] |
    ↳ ::Combine($PSScriptRoot, "simplePod.yaml"), '-o', 'json') |
    ↳ ConvertFrom-Json
219
220 # Do your checks in order to determine whether your tasks have
    ↳ finished or not!
221 # This one checks the kubectl deployments. It will continue when all
    ↳ scheduled resources from the deployment are available
222 Write-VBox4PwshLog -Component $Component -Message ("Waiting now for
    ↳ the changes to be applied") -Level Verbose
223
224 $Pods = Wait-PodDeployment -File
    ↳ ([System.IO.Path]::Combine($PSScriptRoot, "simplePod.yaml"))
225
226 # This hashtable will be used later for exchanging strings within the
    ↳ Action Evidences
227 $StringsToReplace = Find-StringsToReplaceInPods -Pods $Pods
228
229 Write-VBox4PwshLog -Component $Component -Message ("All done!
    ↳ Exporting Image now and create the iDiff(s)")
230
231 foreach($vm in $VMs.Name) { Invoke-SyncOnVM -VM $vm -UserName
    ↳ $VMUserName -Password $VMUserPassword -EnvironmentVariables
    ↳ $VMEEnvironmentVariables}
232 Start-Sleep -Seconds 10

```

Listing 5.2.: Contents of the Analysis.ps1 file of the scenario: Deploy pod

The full script file can be found in Appendix A.5.1. When the file as shown in Listing 5.1 gets applied to the cluster, the *Pod*, the container and corresponding *storage volumes* receive a *Globally unique identifier (GUID)*. As this identifier changes whenever the action is repeated, the formatted output of `idifference2.py` and the `Get-FSMounts` commandlet needs to be searched for the identifier and replaced by a static string. The commandlet `Find-StringsToReplaceInPods` is taking care of that action. Deploying the *Pod* specification returns an output in JSON format (see line 218 in Listing 5.1) which can be parsed directly in *PowerShell*. This output will be parsed in order to search for *GUIDs* and match them with the corresponding object they correlate to.

```
1 # $StringsToReplace | Format-List
2 Name   : f6221c87-927f-4128-ba55-5d23022d3f8f
3 Value  : $PodID$simple-pod$
4
5 Name   :
6   ↪ 7ef8a9bee9996e25f3056493d5e7fb7afa6394fd189eeda30f18ee9f2f824ea4
7 Value  : $ContID$simple-pod-container$
8
9 Name   : kube-api-access-n6jlt
10 Value  : $ContID$VolumeMount$simple-pod-container$1$
```

Listing 5.3.: Overview of replaced strings

The variable is of type `Hashtable` and consists of key-value pairs, represented in Listing 5.3 as name-value pairs. The names are the strings that were replaced with the corresponding value.

The commandlets `Get-CharacteristicEvidence` and `Get-FSMountCharacteristicEvidences` were executed with the specified parameter `-MinCount` which is set to 16. This declares that out of 20 runs an evidence has to be present 16 times to be seen as characteristic. As the count of the evidences is included in the resulting files, it can be determined how often an evidence is present in the end. Furthermore, a parameter will be added to `Invoke-ActionStateFileExport` when the final round is reached which tells the function to not delete the exported raw image files after completion. That allows to exhibit the image of the final round for a post-mortem analysis. Alternatively, the snapshots can be restored so that the action can get performed again. As *Vagrant* enables an ssh connection to both master and worker node, a live analysis could be done.

5.2.3. Results

On the *Ubuntu* analysis machine, the process took around 8 hours. For the master node, there were 7 characteristic evidences found as a result of the file system analyses whereas there are zero characteristic evidences for the file system mounts analysis. This can be explained with having most of the workload assigned to the worker node, where 538 characteristic evidences from the file system analysis and 4 characteristic evidences from the

file system mounts analysis were resulting. The process on the *Windows* machine took approximately 10 hours and resulted in 5 characteristic evidences for the file system analysis and zero for the file system mount analysis on the master node. On the worker node, there were 541 characteristic evidences for the file system analysis and only 2 characteristic evidences for the file system mount analysis. The results from the *Ubuntu* machine can be found in Appendix A.5.3 and the ones from the *Windows* machine in Appendix A.5.4.

The first look will be taken into the file system mount evidences. Both results contain the following lines:

```

1 /run/containerd/io.containerd.runtime.v2.task/k8s.io/_
  ↳ $ContID$simple-pod-container$/rootfs ADDED
  ↳ 20
2 /var/lib/kubelet/pods/$PodID$simple-pod$/volumes/_
  ↳ kubernetes.io~projected/_
  ↳ $ContID$VolumeMount$simple-pod-container$1$ ADDED
  ↳ 20

```

Listing 5.4.: Characteristic file system mount evidences from scenario 1

Listing 5.4 contains two placeholder strings which are explained in Listing 5.3. Line 1 reflects the ID and name of the container as specified in Listing 5.1 while number two shows a *storage volume* mount point which got added during the creation. As this evidence is seen in all 20 runs, an arbitrary file containing the file system mounts can be taken from the results in order to see the mount parameters and settings. For this example, the file `kubernetes-worker-1.DeployPod.20.FileSystemMounts.txt` will be used. The target as shown in line 1 in Listing 5.4 is of type `overlay` which is typically used as a file system for containers. Among other mount-specific options, the `lowerdir` and `upperdir` properties are specified. They are both merged together in a single file system where the `upperdir` is overlaying the structure of the `lowerdir` (Linux Kernel Organization, 2021a). The `lowerdir` mentioned within the mount options is `/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/39/fs` while the `upperdir` entry contains the directories `fs` and `work`, both located in `/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/40`. The directories 39 and 40 can also be found within the characteristic evidences of the file system from the worker node. As a matter of fact, the most changes were performed within the directory `var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapsh`

ots containing many created files. That directory also contains a file named `metadata.db` which got modified and changed. To take a deeper look into that file, it has been exported from the image of the last performed round. To do so, the raw image file has been mounted into *Autopsy* (see Section 3.4.6) beforehand so that the file's location could be found by navigating to it. Information about the database's structure can be found within the source code of the *containerd* repository where it is stated, that the database “represents a metadata database backed by a bolt database” (containerd, 2015a, see "containerd/blob/main/metadata/db.go"). It further states that “the database is fully namespaced and stores image, container, namespace, snapshot, and content data while proxying data shared across namespaces to backend datastores for content and snapshots”, which explains the information that can be found within that database. The package used to implement the database is forked and maintained by the *etcd* project which transformed the usage of the original *bbolt* project solely for the *Go* language (io, 2017). The database layout represents a key-value store. It can be inspected by using the very package that is built into *containerd*. It is documented how to install the package and open a database in read-only mode which is an advantage for forensic investigations.

The SHA1 checksum of the exported `metadata.db` file needs to be verified by looking up its checksum in the corresponding `idiff` file of round 20, which is named `kubernetes-worker-1_DeployPod.20.idiff`. Alternatively, the checksum can be retrieved from *Autopsy*, given that the file hashes were calculated. Within the `fileobject` entry for `metadata.db` both MD5 and SHA1 checksums are shown. The SHA1 checksum is `945b998969434aad0e38f0045ea9715fe508d4e9` and did not change on the exported file. To analyze the database, the *bbolt* binary was used within a containerized environment using the *golang* container image in *Docker*.

```
1 # run the docker container
2 docker run --rm -it -v D:\_CASES\DeployPod\Export:/tmp golang
3
4 # install the bbolt package
5 root@534163ac879f:/go# go get go.etcd.io/bbolt/...
6 go: downloading go.etcd.io/bbolt v1.3.6
7 go: downloading golang.org/x/sys v0.0.0-20200923182605-d9f96fdee20d
8
9 # change to mounted volume
10 root@801c2b3ae7ec:/go# cd /tmp/
11
12 # validate the checksum of metadata.db
```

```

13 root@801c2b3ae7ec:/tmp# shasum metadata.db
14 945b998969434aad0e38f0045ea9715fe508d4e9  metadata.db
15
16 # check integrity of the database file
17 root@801c2b3ae7ec:/tmp# bbolt check metadata.db
18 OK
19
20 # print the basic information
21 root@801c2b3ae7ec:/tmp# bbolt info metadata.db
22 Page Size: 4096
23
24 # print the stats of the database
25 root@801c2b3ae7ec:/tmp# bbolt stats metadata.db
26 Aggregate statistics for 1 buckets
27
28 Page count statistics
29     Number of logical branch pages: 1
30     Number of physical branch overflow pages: 0
31     Number of logical leaf pages: 35
32     Number of physical leaf overflow pages: 0
33 Tree statistics
34     Number of keys/value pairs: 385
35     Number of levels in B+tree: 5
36 Page size utilization
37     Bytes allocated for physical branch pages: 4096
38     Bytes actually used for branch data: 300 (7%)
39     Bytes allocated for physical leaf pages: 143360
40     Bytes actually used for leaf data: 23044 (16%)
41 Bucket statistics
42     Total number of buckets: 72
43     Total number on inlined buckets: 39 (54%)
44     Bytes used for inlined buckets: 6018 (26%)

```

Listing 5.5.: Perform analysis on `metadata.db`. Adopted from (io, 2017)

Importing the package as shown in line 8 of Listing 5.5 installs the `bbolt` binary as well, which is used to interact with the database file. An integrity check can be run to confirm that the file does not contain any corrupted information. The commands `info` and `stats` show detailed information about the database. The documentation states that `stats` could also be used to take snapshots before and after an operation to see what exactly has been performed in the database. This method could be an enhancement for

the analyzing process taken in this work as it would implement a more dedicated method to analyze *containerd*-specific actions. The `pages` command prints a list of pages that are part of the database. Each page has its own size and a specific amount of entries which can be retrieved by the corresponding ID. The items - consisting of key-value pairs - reflect properties and settings that are used by the snapshotter, which manages the contents of images (the layers) as well as the bundles that form the file system of a container (see Fig. 3.4). When analyzing the contents of `metadata.db`, it is worth mentioning that the values are represented in hexadecimal. Thus, it is necessary to convert and format them based on the purpose of a key. There are, for example, the keys `createdat` and `updatedat` which are used for several data types that are stored in a hexadecimal binary format represented as `string`, as all values within the database are of type `string`. The source code file `metadata/boltutil/helpers.go` from the *containerd* (2015a) repository shows that the *Go* package `time` is used to transform the representation of *UTC* timestamps into a savable format using the `MarshalBinary()`. The function `ReadTimestamps` within the same source code file implements the method to restore that value to its original data type by using the `UnmarshalBinary` function of the `time` package. This information can be used to restore the timestamps out of `metadata.db`. To see which information can be retrieved for the deployed container out of the `metadata.db`, the pages are searched manually for the identifier of the container. As the identifier set by *Kubernetes* has been replaced with a placeholder, it needs to be retrieved manually from the `Analysis.log` file. Every time a `kubectl apply` was performed during the analysis, the `JSON` output is traced in the logfile. Round 20 has been the last round, meaning that the needed information has been returned by the last call of that command.

Identifier	Value
Pod ID	564f434f-cb55-4385-b874-33fe31ff31db
Container ID	53ecc572b24eca56b957029d0e5be381be1b6f469ea8674718019ff1b70c16ca
Image ID	sha256:e7157b6d7ebbe2cce5eaa8cfe8aa4fa82d173999b9f90a9ec42e57323546c353

Table 5.1.: Original identifiers from last round of analysis

The container ID as represented in Table 5.1 exists on page 26 and 28. The command `bbold page metadata.db 26` displays the entries for that specific page. The one that contains the ID is at index 14. Using this information, the key-value pair can

the file `snapshots/storage/bolt.go` which contains generic information like structures, variables and functions that are implemented in the snapshot driver files like `overlay.go`. Furthermore, the keys of a bucket stored within the database are defined. Those keys are `v1` which describes the storage version, `snapshots`, `parents`, `id`, `parent`, `kind`, `inodes` and `size`. Those values can be seen in several pages of the database, e.g. page 24 (see Fig. 5.2).

```
root@04475fd27320:/tmp# bbolt page metadata.db 24
3Page ID:      24
Page Type:    leaf
Total Size: 4096 bytes
Item Count: 8

"createdat": 010000000ed91fab3b33d5f4edffff
"id": 18
"inodes": 0a
"kind": 03
"labels": <pgid=0,seq=0>
"parent": "k8s.io/40/sha256:a09552c58f95f44444fcb942fd3bef607848479bf89d72437e09be5966e1e30"
"size": 8080ae05
"updatedat": 010000000ed91fab3b33d5f4edffff
```

Figure 5.2.: Contents of page 24 of `metadata.db`

That information may be used to analyze further details and connections between the several snapshots. Having the source code files analyzed, it is possible to state that a `metadata.db` file contains information about the snapshots and file system layers that get mounted into a container.

Generally speaking, the database needs to be parsed in a programmatic approach in order to restore the layout and information about the given storage information of the *containerd* installation. One tool that already took the given code snippets from the *containerd* repository to implement forensic use cases is *container-explorer* by Google (Google, 2021). A deeper look into this tool will be taken in Section 5.5.

Another sub-directory that contains a *bolt* database is `/var/lib/containerd/io.containerd.metadata.v1.bolt/meta.db` which has also been modified in each single run. Searching the *containerd* repository again for entries specific to `meta.db`, the file `server.go` shows up that defines the path to the database and creates it using the function `NewDB` in lines 419-431 (containerd, 2015a, see "services/server/server.go"). This function is defined in the file `metadata/db.go` which has already been mentioned before. Furthermore, it contains the function `Init` which initiates the database with buckets. Those are described within the file `metadata/buckets.go`. This file also includes the database schema in the comments. Based on that explanation, the database contains information about the basic objects used by the *container runtime* like

labels, *images*, *containers*, *snapshots*, *content* and *leases*. Those objects are stored per namespace. To parse information out of this database, the tool *container-explorer* was used later on. Given the schema, this database is storing the metadata information as shown in Fig. 3.3.

Another entry in the characteristic evidences is showing that the directory `/var/lib/containerd/io.containerd.runtime.v2.task/k8s.io/\$ContID\$simple-pod-container\$` has been created in each run, where `\$ContID\$simple-pod-container\$` is reflecting the *container* ID of that particular run. Unfortunately, the directory is empty within the mounted image. This could mean that a task specific to the created *container* has been created in that directory and got deleted right after finishing. The object *Task* (described in Section 3.2.3) is used to create a running process out of a *container* object. Therefore, the existence of that directory could be an indicator of the container's state, meaning that it was running as the directory is present. Taking a look into the parent directory `/var/lib/containerd/io.containerd.runtime.v2.task/k8s.io` in *Autopsy*, it can be seen that there are several other directories which are named after an identifier reflecting container IDs. Those directories were not accessed during the runs.

The location `/var/lib/containerd/io.containerd.grpc.v1.cri/containers/\$ContID\$simple-pod-container\$` describes another characteristic evidence, also created in all 20 runs. This directory contains a file named `status` which has also been created in each run. The file can be retrieved from the exported image and contains a JSON-formatted text:

```

1 {
2     "Version": "v1",
3     "Pid": 15024,
4     "CreatedAt": 1636713127983723088,
5     "StartedAt": 1636713128113911795,
6     "FinishedAt": 0,
7     "ExitCode": 0,
8     "Reason": "",
9     "Message": ""
10 }
```

Listing 5.6.: Content of the file `status`

The properties within that file describe useful information about the status of a con-

tainer. A process ID is given, which seems to be the ID of the system's process that runs the container. A memory dump of the virtual machine (the worker node) would be needed to verify that this ID actually reflects a system process. In addition to that, three timestamps are given: `CreatedAt`, `StartedAt` and `finishedAt`. The format is a *UNIX timestamp* in nanoseconds. Taking the key `StartedAt` and formatting its value 1636713127983723088 to UTC time will result in `Fri 12 November 2021 10:32:07.983 UTC`, which exactly matches the timestamp that can be found in the log file of the analysis when looking into the last run. Therefore, this information can be seen as verified and indeed describes the starting time of the container. The last timestamp `finishedAt` is set to 0. This seems to indicate that the container has not finished and thus is still running. As this file has been stored to the local disk, it could possibly be restored by using file carving techniques once it got deleted.

Furthermore, four characteristic evidences in the form of files have been created within the directory `/var/lib/containerd/io.containerd.content.v1.content/blobs/sha256`. The first of those is the file `34efe68cca33507682b1673c851700ec66839ecf94d19b928176e20d20e02413`. Opening it in *Autopsy* again reveals a JSON structure.

```
1 {
2   "schemaVersion": 2,
3   "mediaType":
4     ↪ "application/vnd.docker.distribution.manifest.v2+json",
5   "config": {
6     "mediaType": "application/vnd.docker.container.image.v1+json",
7     "size": 1456,
8     "digest": "sha256:7138284460ffa3bb6ee087344f5b051468b3f8697e2d1j
9     ↪ 427bacla20c8d168b14"
10  },
11  "layers": [
12    {
13      "mediaType":
14        ↪ "application/vnd.docker.image.rootfs.diff.tar.gzip",
15      "size": 772792,
16      "digest": "sha256:e685c5c858e36338a47c627763b50dfe6035b547f1j
17      ↪ f75f0d39753db71e319016"
18    }
19  ]
20 }
```

Listing 5.7.: Content of the detected file

Based on the layout and properties shown in Listing 5.7, the file seems to store an image manifest (see Section 3.2.2). Given that information, the image name to that digest can be searched in *Docker*. The image configuration file is addressed in line 7 by its digest. That digest is also the name of the file which is the next characteristic evidence in the directory. It is also available in *Autopsy* and - as the `mediatype` property in line 5 describes - has a `JSON` structure. Furthermore, it states that the image is a *Docker* image. Looking into the file, one can see the common properties for an image configuration file as also described in Table 3.1. As much as it reveals about the configuration of an image and how the container would be started, it does not include the image's name or tag. Therefore, a keyword search has been performed in *Autopsy* to see whether the digest of the *configuration* file (as seen in line 7 of Listing 5.7) can be found as an entry in other files on the exported image. Four different files resulted from that search. The first one is the file `/var/log/syslog` which is not included in the characteristic evidences. Many system components log to the `syslog` file, which means that it has also been changed within the *Noise* phase. Line 5527 contains the information that the image `busybox:latest` will be pulled. Pulling an image means that the underlying *container runtime* tries to look up the image's name in the given container registry in order to fetch the index file that guides to the correct manifest. Line 5532 shows the message of an `ImageUpdate` event which is followed by `ImageCreate` that contains the message `Name:docker.io/library/busybox@sha256:e7157b6d7ebb`
`e2cce5eaa8cfe8aa4fa82d173999b9f90a9ec42e57323546c353`. Followed by that, on line 5534 an informational log line appears which tells that an image reference has been returned that points to the digest of the already seen manifest file. This means that the digest starting with `e7157b6d7ebb` returned the correct manifest to use, thus being identified as the index. Taking another look into the `/var/lib/containerd/io.containerd.content.v1.content/blobs/sha256` folder of the extracted image, that file was found and can get inspected. It contains the layout for an index file as described in Section 3.2.2. In sum, it shows ten manifest file digests which differ by the `platform` property. The first entry already shows the manifest to pull for an `amd64` architecture, which in this case is the digest starting with `34efe68cca33`. Taking a look into the image repository of *busybox* in the *Docker* registry (Docker, 2021b) proves that this digest is the identifier for the image with tag `latest` for `linux/amd64` architectures.

In conclusion, the `syslog` file contains the information which is necessary for recon-

TAG		
latest		
Last pushed 7 days ago by doijanky		
docker pull busybox:latest		
DIGEST	OS/ARCH	COMPRESSED SIZE ⓘ
be0b3422c99d	linux/386	718.81 KB
34efe68cca33	linux/amd64	754.68 KB
368f923b2fde	linux/arm/v6	930.23 KB
+7 more...		

Figure 5.3.: Image information from *Docker* image registry

structuring the way of how an image got pulled and where which part of the image has been stored, as *containerd* logs its operations in there.

As it has now been reconstructed how the correct image to use has been fetched, it is worth taking a look into the manifest again in order to check if the image layers also have been stored. Line 13 of Listing 5.7 shows the digest of that layer starting with `e685c5c858e3`. Again, a file named after that digest is located in the same directory. As defined in the `mediaType` property, that file is a compressed `gzip` archive and can therefore be extracted. In order to verify this, the file has been exported using *Autopsy* and expanded by using the command line `tar -xzf e685c5c858e36338a47c627763b50dfe6035b547f1f75f0d39753db71e319016`. As expected, the contents of that archive are shown afterwards exposing the root directory that has been mounted into the started container. This means that the original layers of the containers are available locally on the file system of the worker node. Finally, it is proven to state that all files included in an image pull flow of *containerd* can be found on the local disk. First, it was checked whether there was an index file for the image available in the *Docker* image registry. That file led to the correct manifest file to be used for the given architecture. Within the manifest file the digest of the configuration file and the necessary layers is stated, which also got fetched. The location of those files is important to know as one can rebuild and inspect a complete container. In a scenario where a container got created that delivered malware into the cluster, or maybe the host and the image has been pulled from a private image registry that is not online any longer, the files are still stored on the local disk. An important question to answer is whether files that are created (e.g. through downloads) within a running container will also be stored on the local disk. Given

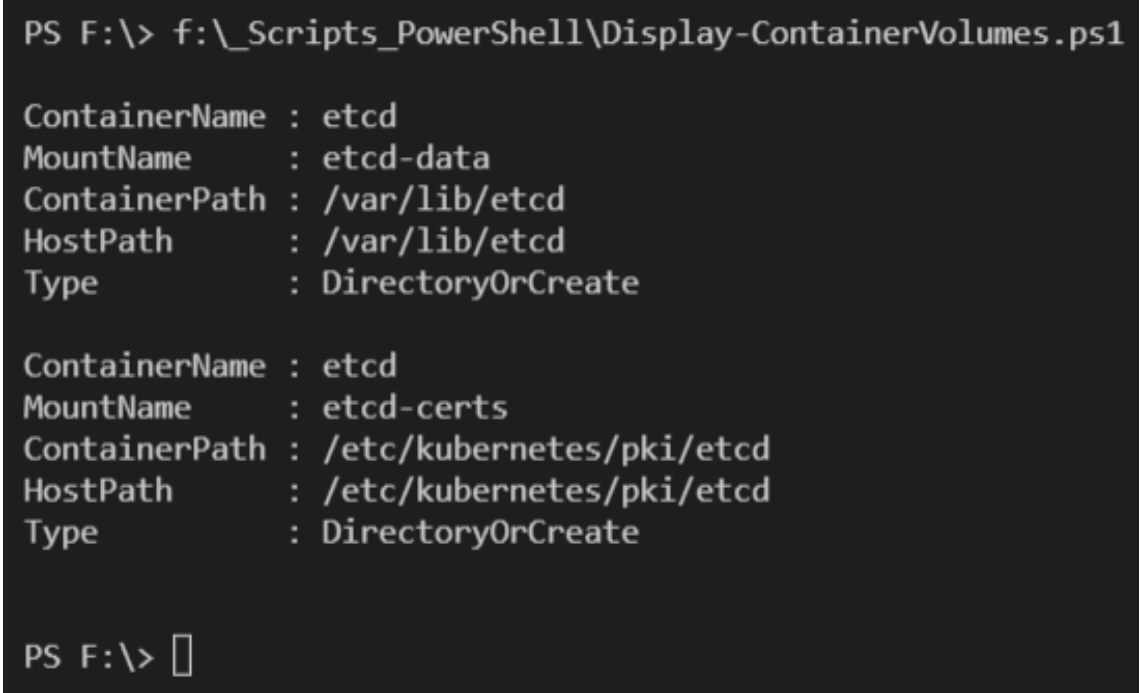
the nature of a container to be ephemeral, this would not normally be expected as the container would start with the file system defined in its manifest when it gets created again. A look into that scenario will be taken in Section 5.3.

The keyword search feature of *Autopsy* has also been used to search for the name and tag of the used image `busybox:latest` in order to check if there are other places where that information is stored. Again the `syslog` file is shown and some entries originating from *containerd* have already been seen. There is another entry at line 5500 that stores multiple objects returned by the *Kubernetes* API such as `PodSpec` and `Container` (Kubernetes, 2021a). Therefore, this relatively long entry contains a lot of valuable information an analyst can use to reconstruct a deployed *Pod* that might have been deleted by the time of the analysis, provided that the information was not deleted. Given that the `syslog` file is a log file, it is very unlikely that an entry from the past gets deleted specifically. On the other hand, a log rotation could happen, where entries are cut when they are older than a given time span. It will be tested in Section 5.4 if this information is still available after a *Pod* gets deleted. The `Container` API object contains the `Command` property which records the invoked command line on the created container. An analyst could find the command line of a malicious container by searching for that specific API object.

Another place where the image name appears is the `meta.db` file, which has already been looked at in the beginning of this section. In addition to that, the several digests seen in the image specification were also found here. The results can be found within the text that *Autopsy* indexed from that file. This proves that the name of the image and its correlation with the digests should also be possible by parsing the `meta.db` database and not only the `syslog` file. On the other hand, an analyst could rely on one of those files in case only either `meta.db` or `syslog` is available.

Furthermore, the term `busybox:latest` appears in the file at path `/var/lib/etcd/member/snap/db` on the master node. This is a sub-directory in the default install location of the distributed key-value store *etcd* (see Section 3.3.1). It is used to store information about the state of a *Kubernetes* cluster and its components and operates as a container deployed within a *Pod* on the master server. In a running *Kubernetes* environment, `kubectl` can be used to list the *Pods* in the `kube-system` namespace, which is the namespace where *Kubernetes* hosts its own components (Kubernetes, 2021k). In a test, the name of the *Pod* was a concatenation of the word `etcd-` and the hostname of the master node. Having the name fetched, further information can be displayed by

invoking the command `kubectl get pods etcd-k8s-m-1 --namespace kube-system -o yaml`. This delivers all information about the *Pod* but, most importantly, the name and ID of scheduled containers as well as mounted volumes. To parse the information, a *PowerShell* script has been written which takes the JSON output and selects the necessary information (see Appendix A.1.4). The following output is produced:

A screenshot of a PowerShell terminal window. The prompt is 'PS F:\>'. The user has entered the command 'f:_Scripts_PowerShell\Display-ContainerVolumes.ps1'. The output shows two sets of container volume information for the 'etcd' container. The first set shows a volume named 'etcd-data' mounted at '/var/lib/etcd' in the container, which corresponds to the host path '/var/lib/etcd'. The second set shows a volume named 'etcd-certs' mounted at '/etc/kubernetes/pki/etcd' in the container, which corresponds to the host path '/etc/kubernetes/pki/etcd'. Both volumes are of type 'DirectoryOrCreate'. The prompt 'PS F:\>' is visible at the bottom of the terminal.

```
PS F:\> f:\_Scripts_PowerShell\Display-ContainerVolumes.ps1

ContainerName : etcd
MountName     : etcd-data
ContainerPath  : /var/lib/etcd
HostPath      : /var/lib/etcd
Type          : DirectoryOrCreate

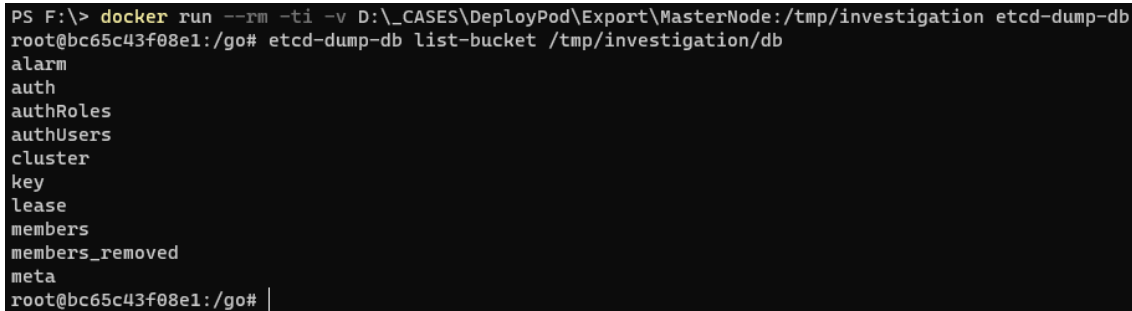
ContainerName : etcd
MountName     : etcd-certs
ContainerPath  : /etc/kubernetes/pki/etcd
HostPath      : /etc/kubernetes/pki/etcd
Type          : DirectoryOrCreate

PS F:\> 
```

Figure 5.4.: Volumes mounted by *etcd* container

This shows that the path `/var/lib/etcd` actually contains the data files from the *etcd* database. To extract the information from that database, the *etcd-io* project offers a tool called *etcd-dump-db*, which can be built manually from the source (etcd, 2013c, see "tools/etcd-dump-db"). For the purpose of this work, the tool has been built into a *Docker* image which can be used to start a container where that tool is already present (see Appendix A.1.5). The exported database file will be mounted into the container's path `/tmp/investigation` where it can now be parsed by using the command `list-bucket` to display its buckets and `iterate-bucket` to iterate over the keys stored in that bucket in reverse order. The terminology is similar to the previously seen database files `metadata.db` and `meta.db` from *containerd*. *etcd* also uses *bolt* in the background and implements the same packages as *containerd* does. A documentation of the implementation and the schema is again delivered through the source code itself

(etcd, 2013c, see "server/storage/schema"). Fig. 5.5 contains a list of buckets that were retrieved by using the tool mentioned before.



```
PS F:\> docker run --rm -ti -v D:\_CASES\DeployPod\Export\MasterNode:/tmp/investigation etcd-dump-db
root@bc65c43f08e1:/go# etcd-dump-db list-bucket /tmp/investigation/db
alarm
auth
authRoles
authUsers
cluster
key
lease
members
members_removed
meta
root@bc65c43f08e1:/go# |
```

Figure 5.5.: List of buckets in *etcd* database

The `iterate-bucket` will now be used to check what information is stored in those buckets. Each bucket is represented as a *Golang* file in the aforementioned repository. In each of those files, API requests are performed to gather data to be stored in the respective bucket, therefore, the official API reference can be used to search for explanations of the buckets (etcd, 2013a). The `alarm` bucket contains alerts that can be triggered by multiple sources. In this case, the bucket is empty. Within `clusters`, the cluster version of the *etcd* cluster is recorded. Both `authUsers` and `authRoles` are empty as well, while `auth` stores a revision item. Metadata information about the database itself is stored in the `meta` bucket. Nodes that are part of the *etcd* cluster are stored within the `member` bucket. In this scenario, only the master node is a member of that cluster. Among the rest of the buckets, `key` contains the most information. Initially, the database file has been found by searching the term `busybox:latest` in *Autopsy*. To search for that term in the `key` bucket, the output will be redirected to a file and searched for manually. Within that output, there are two matches for that term where one of them also matches the layout that has already been seen in the `syslog` file, thus containing the whole information about the deployed pod, including the metadata. This proves that *etcd* stores a complete dump of the *Kubernetes* cluster, which could also be used to rebuild the cluster in case of an emergency. For an analyst, this information is also crucial because the structure of a whole *Kubernetes* cluster could be reconstructed based on the information of that particular database file only. This would, of course, not include actual containers and their layers but at least the metadata of what objects have been deployed where. Hence, next to the directory structure of *containerd* and the entries within the `syslog` file, the database of *etcd* delivers another resource for analysts to get an insight into the *Kubernetes* cluster.

Another interesting characteristic evidence detected on the worker node also seems to store information about *Kubernetes* objects, namely *Pods*. The directory `|/var/lib/kubelet/pods/\$PodID\$simple-pod\$|` got created in each run containing several sub-directories. A part of the path got replaced by the placeholder string of the *Pod* identifier (see Listing 5.3). The file `etc-hosts` is directly located in that directory and contains a `hosts` file. It contains IP addresses and their corresponding hostnames. To check if that file is used in other places, the full qualified path has been searched for in the keyword search of *Autopsy* and an entry has been found in the file `var/lib/containerd/io.containerd.metadata.v1.bolt/meta.db`, the metadata database of *containerd*. The string is encapsulated in a JSON layout which could be extracted as it is a readable string and saved into an external file for further analysis (see Appendix A.5.5). The layout matches the *runtime-spec* configuration layout as defined by the *OCI* (OCI, 2015b) and holds the information that is passed to the low-level *container runtime* which starts the container. Lines 251 to 271, for example, contain the information which *Linux* namespaces are used and where they are located on the host system. The `ipc` namespace is used and located at path `/proc/14975/ns/ipc` as stated in line 258. The string that was searched for initially is part of the *mounts* configuration. The documentation states that those mounts are specified additionally to the *root* mount. On *Linux* hosts, the host's binary mount is used to perform the mount operations. As seen in lines 169 to 178, the file `etc-hosts` has been mounted into the container's path `/etc/hosts`, becoming the `hosts` file of the container. By finding that reference in the `meta.db` file, it seems that the database also stores the *runtime-spec* configuration files for the containers. Once again it should be a requirement for a database parsing tool to also extract that specific information. The information about the used namespaces also includes the hint about the process ID that has been used to run the container. This needs to be verified by taking a memory dump of the running system and comparing the process tables to the entries in *meta.db*.

Within the `containers` sub-directory of path `/var/lib/kubelet/pods/\$PodID\$simple-pod\$`, another sub-directory can be found which has the same name as specified for the container (see Listing 5.1). The directory again contains a file named after a randomized identifier. Only the parent directory has gotten an entry in the characteristic evidences with a count of 20 - the file located in that directory has not. Within the mounted image of the last round, the file got named `2c564ff6` and does not contain any entries but is also shown within the mount configurations of the *runtime-spec* file that has been exported previously. The configuration is shown in lines 179 to 188

and the destination path in the container is `/dev/termination-log`. Information about that particular file path on containers can be found within the documentation of *Kubernetes* where it is stated that the file collects information about the termination of a container. This information can be used for debugging purposes (Kubernetes, 2021j). Given the mount options `rbind` and `rw`, the file on the host path would sync with the file in the container. This enables to also check for termination information after the container has already been stopped or deleted.

An additional sub-directory indicated as characteristic evidence is `plugins/kubernetes.io~empty-dir/wrapped_\[ContID\[VolumeMount\[simple-pod-container\[1\[` which contains the file `ready`. The last part of that directory is a replacement string of a *storage volume* mount identifier specific to the *Pod*. In the round that is analyzed, the replaced original value is `kube-api-access-m4k5q`. Both the directories and the file were created in each round. Again, the file does not contain any entries and therefore has a size of zero bytes. A keyword search for the file's path has been conducted but no results were shown. In a second keyword search, the replaced original value has been taken and the search delivered more results. An entry for `kube-api-access-m4k5q` was found again in the file `meta.db` in the same enclosed JSON string which contains the *runtime-spec* configuration (see Appendix A.5.5). As with the previously mentioned entries, the string `kube-api-access-m4k5q` is a part of a mount configuration in lines 220 to 226. The complete source path is `/var/lib/kubelet/pods/564f434f-cb55-4385-b874-33fe31ff31db/volumes/kubernetes.io~projected/kube-api-access-m4k5q` and got mounted into the target path `/var/run/secrets/kubernetes.io/serviceaccount` in the container of the *Pod*. The *ServiceAccount* object is used to authenticate API requests and gets created by the *kube-apiserver* who will directly assign it to a *Pod*. The object itself contains a signed bearer token that is used for the authentication (Kubernetes, 2021c). The path located on the host or respectively the exported raw image does not contain any further contents. However, when the worker node is up and running, contents are present.

The actual value of the token can be retrieved when logged in to the worker node via SSH. The content in path `/var/run/secrets/kubernetes.io/serviceaccount` in the container deployed by the *Pod* is the same, which has been expected as it is a mounted volume. Taking a look back again into the fetched characteristic file system mounts of the worker node (see Listing 5.4) reveals that the directory is mounted on the worker node. As a full dump of the file system mounts was exported, a deeper

5.3. Scenario 2: Run a specially crafted container

5.3.1. Description

The results of the previous scenario revealed that the contents of a container get stored locally on the respective worker node, at least when it comes to the layers that a container image is using. Given the ephemeral nature of containers, the original contents of an image would just get mounted again when a container gets recreated, meaning that files that were created during the runtime will not survive. The main objective of this scenario was to see if manually created files can also be found on the local disk of a node and if they can be retrieved via a file system analysis. A manually crafted container image was created to perform this operation. The instructions to build the image are specific to *Docker* and are collected in a so-called `Dockerfile`, which can be found in Appendix A.6.2. That file is used by *Docker* to build a container image locally using `docker build`. Once the build process has finished, the image needs to get published to a registry that is also available to *Kubernetes* in order to pull it respectively. *Docker* offers a free plan to use their public container image registry to upload images to a personal repository in that registry. Once the image got pushed from a local device to that registry, it gets publicly available and can get pulled again.

Four sample files were used within this scenario, which have been created manually. Having the control over the sample files allows to track them across a file system and also validate their checksums after an operation has taken place. Additionally, the files are filled with globally unique identifiers to a defined size. This allows to search for the file's contents as well by using the identifiers, e.g. a keyword search in *Autopsy*, or by using file carving methods.

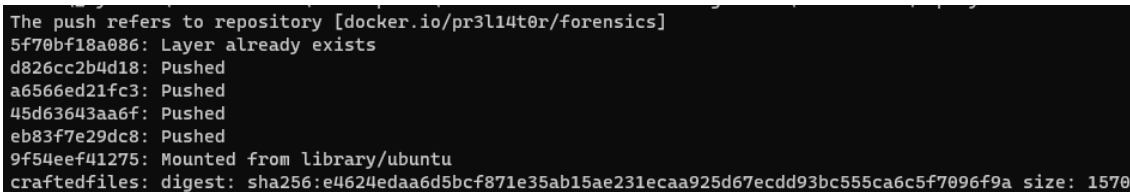
Filename	SHA1-Sum
File01.2MB.tmp	AAFFDFFC47DBF74E3B24D7FBB27ADBF28560601A
File02.2MB.tmp	45DDCDF65324A79A6FF9662FF4708C0582301A96
File03.2MB.tmp	037C75E2588A8E6E44571D786581A7AF3EFCAE5F
File04.2MB.tmp	12DC198BF27A6DBCD1F768E327420D5141413E8F

Table 5.2.: Overview of custom files

File one and two as seen in Table 5.2 will be stored directly in the container image. File two and three were downloaded from the internet by the container after it got deployed.

Following the instructions in the `Dockerfile`, one will notice that the base image is `ubuntu:latest`. That is a rather large image to use as base, but also a well-known and documented one. Line 4 in the `Dockerfile` installed the `wget` package to the image, which has been used to download the files from the internet during the analysis process. Furthermore, the `/home/SampleFiles` directory has been created which stores the sample files. Line 6, 7 and 8 each perform the `ADD` instruction, which copies a file from a local directory into the image. File one and two were copied into that directory as well as a shell script that contains the instructions to download the file. Each of those operations creates another layer that will be used by the image. Line 10 then defines the `WORKDIR` which will be the default directory of a container that runs the image. Lastly, a `CMD` instruction is set in line 12. If not specified otherwise when starting a container using that image, the `/bin/bash` binary will be executed.

The image has been built locally and named `pr3l14t0r/forensics:craftedfiles`. Using that image name, it can now be used by *Kubernetes* and get pulled by *containerd*. The `docker push` command also shows the new layers and their digests that have been added to the image manifest.



```
The push refers to repository [docker.io/pr3l14t0r/forensics]
5f70bf18a086: Layer already exists
d826cc2b4d18: Pushed
a6566ed21fc3: Pushed
45d63643aa6f: Pushed
eb83f7e29dc8: Pushed
9f54eef41275: Mounted from library/ubuntu
craftedfiles: digest: sha256:e4624edaa6d5bcf871e35ab15ae231ecaa925d67ecdd93bc555ca6c5f7096f9a size: 1570
```

Figure 5.7.: Pushing an image in *Docker*

In the last line seen in Fig. 5.7, the tag *craftedfiles* is shown with its corresponding SHA256 digest.

5.3.2. Analysis

The *Init* phase will start off from a freshly deployed *Kubernetes* cluster. A *Pod* resource will get applied which is specified in the file `craftedPod.yaml` (see Appendix A.6.3). The *Pod* will instruct *containerd* to pull and run the named image. Once the container has been started, the missing files three and four will be downloaded from the Internet. To do so, the script `DownloadFiles.sh` (see Appendix A.6.4) will be invoked. The script is already part of the image and therefore available without having to copy it manually to the running container. In order to run the script on the container, a *Pod* execution

command for `kubectl` will be used. As only one container is scheduled within the *Pod*, no dedicated container needs to be specified to run the command on.

```

236 # Execute the script on pod and fetch return value
237 # 'kubectl exec' should be run with '-it' so that everything will be
    ↳ redirected to STDOUT.
238 # see: https://stackoverflow.com/questions/69393723/kubectl-exec-out_
    ↳ put-not-getting-stored-in-variable
239 $ret = Invoke-Kubectl -kubectlArgs
    ↳ ("exec", "craftedfiles-pod", "-it", "--", "./DownloadFiles.sh")
240
241 # Do your checks in order to determine whether your tasks have
    ↳ finished or not!
242 # This one checks if a certain line has been returned from the
    ↳ script
243 if($ret.Contains("Finished download and verified the
    ↳ files!")) {Write-VBox4PwshLog -Component $Component -Message
    ↳ ("Successfull!")}
244 else {Write-VBox4PwshLog -Component $Component -Message ("NOT
    ↳ succesfull! Please check logfile...")}

```

Listing 5.8.: Download of files during the *Action*-phase

In line 239 of Listing 5.8, a helper function parses the needed parameters to `kubectl` which invokes the script on the container. The output of the shell script is fetched in a variable and checked for the success state of the operation. A message will be displayed to console in case the download operation was not successful. All other steps like fetching the file system mounts, exporting and analyzing the disk images, etc., follow the standard process. The count filter for characteristic evidences is again set to 16 and will be applied at the end. The whole analysis was executed on both the *Windows* and *Ubuntu* machines. In order to be able to analyze the file system after the *Action* phase has finished in a post-mortem approach, the disk exports from the last round taken were kept, but only from the *Windows* machine. They have been imported into *Autopsy* to perform further analysis.

5.3.3. Results

The analysis process took 0:07:44:17h on the *Ubuntu* machine. The file system analysis produced 3 characteristic evidences on the master node and 19 on the worker node. For

the mount analysis, there were only 4 characteristic evidences on the worker node and none on the master node. On *Windows* the process took 0:10:38:33h and for both the file system and the mount analysis, no characteristic evidences were found on the master node. On the worker node, there were 11 characteristic evidences for the file system analysis and 4 results for the mount analysis. The mentioned result files can be found in Appendix A.6.5 and Appendix A.6.6.

The analysis on *Ubuntu* and *Windows* resulted in different evidences for the master node. Therefore, a look into the file system evidences for the master node has been taken first. There were only 3 entries with a counter lower than 18.

```
1 $OrphanFiles/OrphanFile-5373966 d 17
2 tmp/ansible_command_payload_ygyp7pcl d 18
3 tmp/cri-containerd.apparmor.d046889364 d 17
```

Listing 5.9.: Characteristic file system evidences detected on master node

All of the files shown in Listing 5.9 were deleted during some of the rounds and do not seem to be related to the action that has been taken. The file in line 2 refers to an *Ansible* command payload that was used in the phase of provisioning the virtual machine that hosted the master node after the creation with *Vagrant*. Based on its naming, the file mentioned in line 3 was used by the `AppArmor` package installed on the virtual machine. Unfortunately, no further information on that file was found. Given that the files do not have a count of 20 and seem to be generic, they will be considered as false positives and therefore not seen as characteristic evidence. That would conclude that no characteristic evidences for the master node were found on both the *Windows* and *Ubuntu* analysis machines.

The first entry both result sets have in common is the file `resolv.conf` which got accessed in each round. The file is stored in a directory named with an identifier at path `/var/lib/containerd/io.containerd.grpc.v1.cri/sandboxes`. This path hosts information for the so-called *sandbox containers*. Each *Kubernetes Pod* contains a sandbox container which just keeps running forever. It is actually used to create the environment for a *Pod*, e.g. creating and maintaining *Linux* namespaces and other resources that are needed to encapsulate the *Pod*. Technically speaking, the sandbox container process on the host machine is the one that starts other containers as process via system calls (see Section 3.1). This allows to keep a *Pod* available even when the

scheduled container has already finished running (Kubernetes, 2016). The `resolv.conf` file gets mounted into that sandbox container in order to provide basic networking functionality to the *Pod* environment.

The second characteristic evidence that has been seen on both analysis environments are entries within the directory `/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/45`. It was already detected in the previous analysis that the path `/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs` hosts the snapshots of image layers that get mounted into the container. Taking a look at line 17 within the file that stores the mount points from round 20 proves that the sub-directories with name 39 to 44 were used as `lowerdir` while 45 is used as `upperdir`, thus being the active snapshot and therefore the container's root file system. Fig. 5.7 shows that 6 layers have been pushed to the *Docker* image registry and also shows the digest that addresses the built image. As a result of the last analysis, it is now known that this digest reflects the image's manifest file and that *containerd* stores it in the path `/var/lib/containerd/io.containerd.content.v1.content/blobs/sha256`. Opening that directory in *Autopsy* reveals that a file named after this digest is present, containing the JSON-formatted data of the manifest (see Appendix A.6.7). The manifest contains the property `config`, which reveals where the image configuration file is stored, and the property `layers`, which contains information about all layers that are used by the container image. The values stored in that property reflect the layers that have been pushed to the registry as seen in Fig. 5.7, even though the digests do not match at first sight. The layers stored in the *manifest* contain a property called `mediaType`, which is set to `application/vnd.docker.image.rootfs.diff.tar.gzip` for all layers in this example. That describes the layers as zipped archives and thus the SHA256 checksum of the layer becomes the digest. If the layer gets uncompressed using `gunzip`, the checksum and therefore the digest will change.

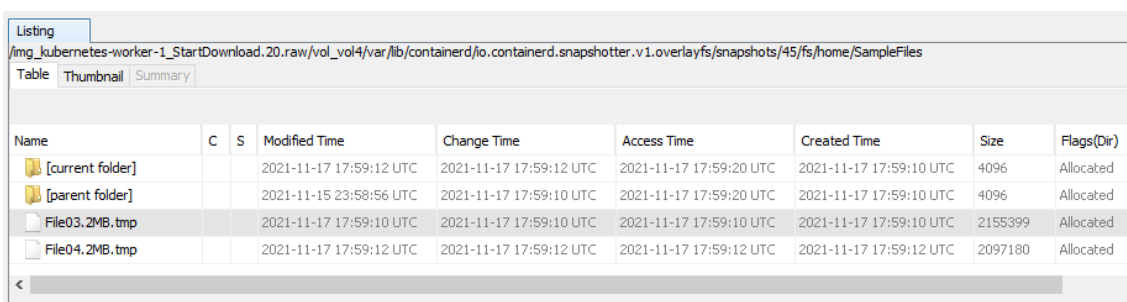
```
> cat 7b1a6ab2e44dbac178598dabe7cff59bd67233dba0b27e4fbd1f9d4b3c877a54 | gunzip - | sha256sum -
9f54eef412758095c8079ac465d494a2872e02e90bf1fb5f12a1641c0d1bb78b -
> |
```

Figure 5.8.: New checksum after extraction of a layer

This has been tested in Fig. 5.8. The first mentioned layer in the manifest has been exported and uncompressed with `gunzip`. The checksum of that uncompressed layer reflects the digest of the first layer as seen in Fig. 5.7, or, to state it in another way: The image's configuration file labels the layers using the checksum of their uncompressed state while the manifest file uses the checksum after a compression of the layers as the

digest. The `mediaType` property in the manifest file reveals which compressing tool has been used for the operation. The uncompressed layers will be mounted into the running container (containerd, 2015f).

As described in Section 5.3.1, the first two of the crafted files are already part of the manually created image and should therefore already be in the root file system that is mounted into the container which has been created in the *Init* phase. Running the provided script, the remaining two files (see Table 5.2) should get downloaded during runtime. Those two files are indeed now part of the characteristic evidences. They are located at path `/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/45/fs/home/SampleFiles/`. The first part of that path (until `../45/fs`) is the `upperDir` mounted into the `rootfs` of the container. The last part therefore reflects the relative path on the container itself. That is exactly the path that has been created in the `Dockerfile`. This shows that files which are created during the runtime of the container actually get stored on node system level. The files can be exported normally from *Autopsy* in order to verify their SHA1 checksums.



The screenshot shows the Autopsy interface with a file listing table. The table has columns for Name, C, S, Modified Time, Change Time, Access Time, Created Time, Size, and Flags(Dir). The files listed are [current folder], [parent folder], File03.2MB.tmp, and File04.2MB.tmp. The File03.2MB.tmp and File04.2MB.tmp files are highlighted in grey.

Name	C	S	Modified Time	Change Time	Access Time	Created Time	Size	Flags(Dir)
[current folder]			2021-11-17 17:59:12 UTC	2021-11-17 17:59:12 UTC	2021-11-17 17:59:20 UTC	2021-11-17 17:59:10 UTC	4096	Allocated
[parent folder]			2021-11-15 23:58:56 UTC	2021-11-17 17:59:10 UTC	2021-11-17 17:59:20 UTC	2021-11-17 17:59:10 UTC	4096	Allocated
File03.2MB.tmp			2021-11-17 17:59:10 UTC	2021-11-17 17:59:10 UTC	2021-11-17 17:59:10 UTC	2021-11-17 17:59:10 UTC	2155399	Allocated
File04.2MB.tmp			2021-11-17 17:59:12 UTC	2021-11-17 17:59:12 UTC	2021-11-17 17:59:12 UTC	2021-11-17 17:59:12 UTC	2097180	Allocated

Figure 5.9.: Manually downloaded files shown in *Autopsy*

The checksum of the exported files corresponds to the ones calculated beforehand (see Table 5.2). This means that the original files are available on the host system, as long as the container's file system is mounted. It needs to be checked what happens to the downloaded files when the *Pod* resource gets destroyed. This will be analyzed in Section 5.4.

To check if the file names are occurring in other locations or files, a keyword search for the file names has been conducted. For all files, only the file itself has been found in the already named location of the snapshotter. For file three and four, an entry has been found within the `DownloadFiles.sh` file which is part of the container's image. The keywords are part of the hardcoded output that the script delivers.

As already seen in the evidences of scenario 1 as well, the log files at paths `/var/log/containers` and `/var/log/pods` have been accessed. As the container was

already running in the *Init* state, the log files were present in the *Action* phase and thus only accessed. On the other hand, this also means that the output produced by the shell script has not been written to `/var/log/pods/default_craftedfiles-pod_\
$PodID\craftedfiles-pod\`, which only shows the output produced when the *Pod* got initially created.

To check whether the output delivered by the executed script has been logged somewhere else, a keyword search has been performed once again using some hardcoded output strings from the script as search value. The only hit in that search was the shell script itself, which means that no further locations storing the output are known.

Another keyword search has been done for the name of the script file that has been invoked to download file three and four in order to find out if the action is logged anywhere. This search had a hit for the already seen file `/var/log/syslog`.

```
1 Nov 17 17:59:08 ubuntu2004 containerd[8951]:  
  ↳ time="2021-11-17T17:59:08.205358793Z" level=info msg="Exec for  
  ↳ \"2df8c9bf881593eea104ed0cce6234582aef7f26a13fdf034ca1fdd6abf60b4_]  
  ↳ 6\" with command [./DownloadFiles.sh], tty true and stdin  
  ↳ true"
```

Listing 5.10.: Logged entry in `syslog`

The logged entry contains detailed information about the executed command and the container the command has been sent to. This also helps in a post-mortem forensic investigation to reconstruct possible executed attacks on containers.

5.4. Scenario 3: Deletion of a *Pod*

5.4.1. Description

The objection of this analysis scenario was to find out what traces are left when a *Pod* gets deleted. Furthermore, a look into the created files and folders from the previous two scenarios will be taken to see if created items will still be found on the systems. The container scheduled by the *Pod* in the previous scenario contains manually crafted files, which gives a good base to search for the files as the checksums can be generated to use them for *Autopsys* hashset lists. Plus, the file's content is filled with a unique and repeating string in the form of a GUID. This information can also be used to search for the deleted files. Besides that, a look into the state of the characteristic evidences seen in Section 5.3.3 will be taken to find out if any persisted even after the deletion of the *Pod*.

5.4.2. Analysis

The *Action* state from scenario 2 describes the *Init* state for this scenario. Nonetheless, the cluster has been redeployed to work with a fresh environment. To bring the cluster in its *Init* state, the *Pod* as defined in Section 5.3.1 has been deployed again and right afterwards the script on the container was executed using `kubectl`, also reusing the command from the previous scenario. In the *Action* phase, a simple `kubectl delete` command will be invoked, parsing the *Pods* definition file as input so that *Kubernetes* knows which *Pod* to remove. Passing the path to the *Pod* definition file brings in the advantage of not hardcoding the *Pods* name into the analysis script, thus increasing the usability.

```
248 # Deleting the pod here
249 $ret = Invoke-Kubectl -kubectlArgs ("delete", "-f", [System.IO.Path]::C\
    ↳ ombine($PSScriptRoot, "craftedPod.yaml"), "-o", 'name') -RetryCount
    ↳ 30
250
251 # Do your checks in order to determine whether your tasks have
    ↳ finished or not!
252 # This one checks if a certain line has been returned from the
    ↳ script
253 if($ret -like "pod*deleted*") {Write-VBox4PwshLog -Component
    ↳ $Component -Message ("Successful!") }
```

254

```
else {Write-VBox4PwshLog -Component $Component -Message ("NOT
↪  succesfull! Please check logfile...") }
```

Listing 5.11.: Command that deletes the *Pod*

The output of the `kubectl` command in line 249 is fetched and checked in line 253. If the returned string contains the *Pod*'s name and the term *deleted*, the operation is seen as successful. As also done in the prior analysis scenarios, the occurrence counter for characteristic evidences will be set to 16. The exported raw disk images from the last round will be kept so that they can be imported into *Autopsy*.

5.4.3. Results

The analysis process took 0:07:59:17h on the *Ubuntu* machine. The file system analysis produced 8 characteristic evidences on the master and 123 on the worker node. For the mount analysis, there were 5 characteristic evidences on the worker node and none on the master node. The process took 0:12:26:15h on *Windows*. 65 characteristic evidences have been found on the master node for the file system analysis and zero for the mount analysis. On the worker node, there were 100 characteristic evidences for the file system analysis and 5 results for the mount analysis. The mentioned result files can be found in Appendix A.7.3 and Appendix A.7.4.

To perform the post-mortem analysis *Autopsy*, the exported disk images from round 20 of the *Windows* machines were imported. The necessary identifiers needed for the analysis can be fetched from the `AnalysisLog.txt` file, as the JSON response from the *Pod* deployment from the *Init* phase has been logged. Alternatively, the `/var/log/syslog` file may be considered for obtaining the identifier information, as the commands and requests from *Kubernetes* and *containerd* are logged there as well.

Taking a look into the characteristic evidences, one can see that many items were deleted during the operation. The directory named after the *Pod*'s ID in `/var/lib/kubernet/pods` has been deleted recursively. It is not available on the exported image of the worker node. Furthermore, several items in `/var/lib/containerd` have been deleted as well. The directory `io.containerd.grpc.v1.cri/containers/\$ContID\$craftedfiles-container\$` still exists, even though the container should have been deleted. It contains the file `status` which is allocated. This file has

also been seen in the results of scenario 1 and 2 in the corresponding paths. Included in that file is JSON-structured information like creation and start time. The field `FinishedAt` also includes a value this time, which resolves to Mon 22 November 2021 12:06:45.801 UTC. It describes the day the container was stopped. The file furthermore contains an `ExitCode` which is set to 137 while the fields `Reason` and `Message` are empty. The `ExitCode` indicates that the process has been terminated by `SIGKILL`, which seems to be sent when the *Pod* resource was deleted. As described in the results of scenario 2, every *Pod* environment is created by a sandbox container. To get the identifier of that container, a look into the `syslog` file from the exported image has been taken. Amongst several other events that transcribe the deletion of resources, there is a line that contains the identifier of the sandbox container that ran the *Pod*. Using this identifier to take a look into the sub-directory which is named after the identifier in `/var/lib/containerd/io.containerd.grpc.v1.cri/sandboxes` reveals that the directory also still exists. It also still contains the file `hostname` which stores the hostname of the *Pod*. Using that information, it is possible to find out the name of a deleted pod. The directory further contains the files `resolv.conf` and `hosts` which have also been seen and explained in Section 5.3.3.

Furthermore, it is interesting that no characteristic deletions are shown for both the content storage where the specific image files are stored and the snapshotter directory, which stores the layers that were mounted into an image. This is where the hash lookup plugin from *Autopsy* comes into play as well. Both the files three and four (see Table 5.2) were identified by their MD5 checksum in the sub-directory `snapshots/45/fs/home/SampleFiles` of the *overlayfs* snapshotter. This concludes that an active snapshot does not get deleted even though the container the snapshot has been mounted into does not exist any longer. So, if that container would have downloaded a malicious binary, it could still be retrieved from the directory of the container's active snapshot, even though the container has been deleted. It could be explained with the situation that a *Pod* can be configured to run multiple instances of the same container. Keeping the active snapshot allows a duplicated container to work in case another one is broken. The observed behaviour is unexpected because only one single container has been scheduled by the *Pod*.

Another location where no files have been deleted in a characteristic way is the content storage location at `/var/lib/containerd/io.containerd.content.v1.content/blobs/sha256`. The previous analysis revealed that this place contains the files related to the *OCI* image specification, namely a container image's index, manifest and configuration as well as the layers used within that image. This is still the case within

the current scenario. In scenario 1 the `syslog` file, for example, revealed the digest to an index file of a container image when a pull operation is performed. This would help an analyst for the process of recovering a container. If a malicious container image was downloaded that is not publicly available any longer, the corresponding items could be retrieved via the content storage location, allowing to rebuild the file system of a container out of the stored layers. Furthermore, the configuration file would contain the standard process that is started on the container. If the command was changed or another command was executed on an already running container, the `syslog` file would contain sufficient information.

In addition to the characteristic evidences on the file system, the information of the metadata store and the snapshotter database will get examined. To do so, the files `meta.db` and `metadata.db` will be exported and parsed with the tool *container-explorer* (see Section 5.5). Using the `list container` option from the tool reveals that there is still an entry for the container that has been scheduled on the *Pod*. The entry contains both the name and tag of the container image as well as the identifier of the container. Furthermore, the name of the *Kubernetes Pod* is stated within the `Labels` field. Concluding that the information about a deleted container is still available in the metadata store of *containerd*, it can be used either complementary to or instead of the information that is also given in the `syslog` file. The fact that the information is available in the database also states the importance of those files both for forensic investigations and for backup scenarios.

Another file that has been analyzed in scenario 1 as well is the database of *etcd*, located on the master node. It will also be exported and analyzed with *etcd-dump-db* to check for information that might still be stored. As also done in Section 5.2.3, the `key` bucket will be iterated and the output will be redirected to a text file. This text file has been searched for the name of the *Pod* and several entries were found. From top to bottom of the given outputs, several lines contained the detailed *Pod* specification, but none of them indicated that the *Pod* was deleted. However, the database still contains snapshots of the *Pod* definition used within this scenario. So, even after the deletion of the *Pod*, its structure can be reconstructed. It is further possible to compare several states of that *Pod* to another as multiple definitions are stored from different points in time.

5.5. Evaluation of container-explorer

Analysing the results in Section 5.2.3 revealed that *bolt* databases are used in several places by both *containerd* and *Kubernetes*. One problem that occurred during the analysis was the difficulty of parsing those databases, especially when nested buckets are used as done for `metadata.db` and `meta.db`. The `db` database used by *etcd* was successfully parsed using the tool *etcd-dump-db*, but this will not work for the databases used by *containerd* as the iterator would only iterate over one bucket, and cannot re-iterate over nested ones. To retrieve more information out of the databases, a tool is needed that implements the database's functions from *containerd* itself, specifically for the sake of forensic investigation. That includes opening the database files in read-only mode and correlating the data from both `meta.db` and `metadata.db`. Furthermore, it should support working in a post-mortem scenario, meaning that the host where *containerd* is installed is not available in a running state, but either a raw disk image or the file system in form of a copy or a mountable snapshot is present.

During the research for such tools and for evaluation whether or not a manual program needs to be developed, *Google* released a tool written in *Golang* that seems to fulfill exactly this requirement. The tool is called *container-explorer* and was released on the 1st of November 2021 (Google, 2021). The description states that an offline *containerd* runtime can be analyzed with that tool by either mounting a disk image or by redirecting to a path that contains the root folder of a *containerd* installation. Further, it is stated that components of a *containerd* installation as seen in Fig. 3.3 can be explored. The tool shall be examined using the resulting image file of Section 5.3 that contains the cluster's state right after the deployment of a *Pod* with a crafted container that downloaded files from the Internet. It will be discussed whether a forensic investigation can be conducted with it.

No public pre-built releases are distributed within the repository of *container-explorer*, which means that the tool needs to be built locally. As already done with the tool *etcd-dump-db*, a *Docker* image has been constructed in which the tool is built during the image creation process and executable when finished. This enables to run the tool in an isolated environment on the analysis machine, given that *Docker* is installed. To work with an exported disk image, the image needs to be mounted in a forensically sound way. This can be achieved by mounting it in read-only mode. A checksum must be calculated beforehand and afterwards. As stated, an exported raw disk from scenario 2 (see Section 5.3) was used for this purpose. The image has been exported from the worker node in round

20 from the *Action* phase. Using the *WSL2* on the *Windows* analysis machine, it has been mounted in read-only mode and can further be mounted into the container that runs the previously created *container-explorer* image. The *Dockerfile*, building instructions and the instructions used to mount the exported disk image can be found in Appendix A.1.6.

The first objects to be listed are the containers. To do so, the path to the mounted volume is being specified in the parameter `--image-root`. This instructs *container-explorer* to look into the standard installation path of *containerd* which is `/var/lib/containerd`. 9 entries were returned in a tabular view containing the following headers: Namespace, Container Name, Image, Created At and Labels. These are the values that have been found in the source code from *containerd* that shows the database schema of `meta.db` in the comments (containerd, 2015a, see "metadata/buckets.go"). Comparing it to the implementation of *container-explorer*, this schema has been adopted in the file `ctrmeta/manifest.go` and therefore extracts the information of the `meta.db` file. This also supports the assumption made in Section 5.2.3 that `meta.db` reflects the metadata store of *containerd* as described in Fig. 3.3. As the raw disk image is still available in *Autopsy*, a direct comparison between the returned output and the local directories for containers can be made. The path `/var/lib/containerd/io.containerd.grpc.v1.cri/containers` contains two sub-directories: `containers` and `sandboxes`. As explained in Section 5.3.3, *Kubernetes* starts so-called sandbox containers that create a *Pod* environment which hosts the containers declared in the *Pod* definition file. The `containers` directory contains 6 sub-directories and the `sandboxes` directory contains 3, which sums up to 9 and matches exactly the returned output of *container-explorer*. The container names as shown by the tool contain the container IDs, which is also the naming convention for the folders in the aforementioned directories. This proves that the tool is not missing out information. The same command to list containers has been invoked again, but instead of specifying the root path to the mounted image, only the path of the `meta.db` file has been specified in parameter `--manifest-file`, which delivered the same results.

Another functionality of the tool is to list namespaces. This can also be achieved by either providing the path to the root directory of the mounted image or to an exported `meta.db` file. Either way returns the list of configured namespaces which, in this case, is the default `k8s.io` one, as no custom namespaces have been defined in *Kubernetes*.

The third option delivered by the tool is to list the content information. Entries for the content store of *containerd*, which reflect the files that are located in the directory `/var`

`/lib/containerd/io.containerd.content.v1.content/blobs/sha256`, are returned by that command. There are 52 items within the directory, which matches the sum of the values returned by *container-explorer*. Properties that are included in the returned table overview of *container-explorer* are Namespace, Digest, Size, Created At and Labels, where the digest matches the local file name in the mentioned directory. Interesting information is included within the field Labels. For some of the digests (files), a distribution source is listed, which tells from where a resource has been fetched. If the output is filtered for the name of the image that was used for the crafted container, the output only shows the corresponding digests and therefore the files that are associated with that image. An example Label entry looks like this: `containerd.io/distribution.source.docker.io=pr3l14t0r/forensics`. This information would be important in a forensic investigation because it states from where an image has been pulled. Hence, `metad.db` is an additional source to the `syslog` file, which also contained that information. Again, both ways of specifying the data source have been tested. The results are the same either way.

Another command and the fourth option of the tool is to list the fetched container images. The command returns a list with the headers Namespace, Name, Created At, Digest and Type. The list contains an overview of an image's name and tag in the Name field as well as the corresponding digest of their index or manifest file. These can be distinguished by the actual type name. The type `application/vnd.docker.distribution.manifest.list.v2+json` identifies a file as an image index while `application/vnd.docker.distribution.manifest.v2+json` identifies a manifest (ibid., see "images/mediatypes.go").

The next command lists the snapshot entries. It cannot be run when only the `--manifest-value` parameter is specified as data source, because the snapshot-related metadata is located in the sub-directories of the snapshotter plugins, like `/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs`, which is the location of the *overlayfs* plugin. To invoke the command, either the root directory of the mounted image or both the `--manifest-file` and `--snapshot-metadata-file` parameter containing the path to the corresponding files must be specified. This file has also been discovered during the analysis of scenario 1 in Section 5.2.3. The returned table contains the following headers: Namespace, Snapshotter, Created At, Kind, Name, Parent and FSPath. The field Snapshotter contains the information about which snapshotter plugin was used for that entry. In Kind, the status of the snapshot is described, which can either be committed or active. Only the active

snapshots can be mounted into a container. The field `Name` contains a three-part string which contains the value `k8s.io` at the beginning, followed by a number and ending with the identifier of a container. For each entry, the `Parent` field shows the parent snapshot which, per definition, has to be in a committed state. The `FSPath` field shows the path of that snapshot on the file system. For the container running the manually crafted image, the value is `snapshots/45/fs`, which reflects the characteristic evidence seen in Section 5.3.2. That directory contains the files that have been downloaded. A further information is that only the active snapshots contain the ID of a container in the name. The `Name` field for committed snapshots contains the SHA256 digest which identifies their compressed layer on the file system. Given the available information, the list is comprehensive enough to determine which layers are mounted into a single container. This underlines the importance to have both the `meta.db` and `metadata.db` databases available for a forensic investigation.

Besides listing *containerd*-related information, *container-explorer* also provides the functionality of mounting a single or all containers identified by their ID. The ID of the container that has been created during scenario 2 has already been identified by listing the containers. This ID is now used to test the mount capabilities of the tool. In order to use the mount capabilities, the container that runs the *container-explorer* images needs to be started with the `--privileged` flag. After that, a mount point has to be created in the container where the dead container from the exported image can be mounted into. As of the time of writing, *container-explorer* is using the `mount` binary in the background to mount an `overlayfs` file system into the provided mount point. By specifying the container ID, the snapshot information will be loaded from the `metadata.db` file in order to retrieve all active and committed snapshots the container was using. When specifying the debug flag, the output shows the options passed to `mount`. The analysis process of scenario 2 stored the live information of file system mounts on the worker node by fetching the output of `findmnt`, which enables to compare the used mount options from *container-explorer* to the ones that were used on the live system. The directories fetched for the `lowerdir` options are the same as used by the live system, except the directory that has been the `upperdir` on the live system, which became a `lowerdir` as well. The `upperdir` used in the command from *container-explorer* is the specified mount path within the container itself, which makes sense as the dead container could and should not be mounted into its former file path that reflects the active snapshot. This comparison proves that the tool is fetching the correct paths of the used snapshots and mounts them as an `overlayfs` in a forensically sound manner by using read-only options.

Unfortunately, the mount path is not showing the actual contents of the container. The reason for that is the tool passing the file system paths to be used as `lowerdir` in the wrong order. This can be fixed by manually reversing the order of `lowerdir` directories. An issue regarding this matter has been opened in the official *Github* repository that also explains the problem in detail ¹.

So, instead of using the inbuilt mount command, one can use the `mount` binary directly in order to mount the used snapshots of a container as `overlayfs`. This allows an analyst to inspect the file system of a container in its last running state.

The fact that the tool is heavily relying on both `meta.db` and `metadata.db` once again shows the importance of those files.

¹<https://github.com/google/container-explorer/issues/3>

SUMMARY

The evaluation of the results produced in the analysis scenarios has shown that there are plenty of locations containing valuable forensic information that can be used in an investigation. Given that some of the identified artifacts have persisted even after the deletion of the corresponding *Kubernetes* resources, the analyst is able to reconstruct the layout of that resource or even a whole cluster. In addition to that, one can rebuild the containers that were used within the cluster through the persisted resources of *containerd*. The identified artifacts will be structured within this chapter and be brought into a machine and human readable format. Layout, format and style will be based on the *Github* repository *ForensicArtifacts/artifacts* (Castle and Metz, 2014). The repository will be forked and the identified artifacts from Chapter 5 will be contributed. The artifacts will be separated into *Kubernetes* and *containerd*-related artifacts. While this thesis was written, a contribution was added to the repository which defines some artifacts for *containerd* already ¹. This information will be used to verify the detected artifacts from this work and might possibly get enhanced.

Starting with *containerd*, the most important information is stored within the database files. All contents of the metadata store from *containerd* are stored in the `meta.db` file. It includes information about all objects of the components listed in the architecture (see Fig. 3.3). It is especially useful to combine and correlate information together, like the digest of an index file to an image name. All listed components have an own directory in the main installation directory of *containerd*, which is `/var/lib/containerd`. The sub-directory `io.containerd.content.v1.content/` stores all files that are related to the *OCI* image specification, like the index, manifest, configuration and the

¹see <https://github.com/ForensicArtifacts/artifacts/pull/439>

layers. Using that information, an analyst can reconstruct a container. Status information about containers can be fetched from the sub-directory `io.containerd.grpc.v1.cri`, which splits into the directories `containers` and `sandboxes`. The `sandboxes` directory stores a directory named after the identifier of the container that hosts a *Pod*. As the analysis of the `syslog` file revealed, a sandbox container gets created when a new *Pod* is requested. Once the sandbox container is running, the container scheduled in the *Pod*'s manifest is getting created. In every of those sub-directories named after the container identifier, a `status` file can be found hosting metadata about the container itself. Another sub-directory of the installation path from *containerd* is `io.containerd.metadata.v1.bolt` which contains the `meta.db` file. Together with the snapshotter database(s), which are hosted in the respective sub-directories `io.containerd.snapshotter.v1.<FSType>`, the information stored in there becomes crucial for an analyst. As already stated, information about all components of *containerd* can be found there. Using tools like *container-explorer* to automatically parse that data allows an analyst to quickly check where the latest active snapshot of a container is stored on the file system in order to retrieve files that have been created during the runtime. This also allows to mount containers during a post-mortem forensic investigation in a sound manner. The differential analysis of file system mount points during the execution of the scenarios revealed that the `rootfs` of a container is mounted into the sub-directory `io.containerd.runtime.v2.task/k8s.io`. The name of that directory, which is located at the installation folder of *containerd*, contains the term `task`, which reflects the actual operation responsible for running a container. During a live analysis, the mount points of a host can be fetched by using `findmnt` which will show the mounted `overlayfs` layers. Those layers are stored in sub-directories of the previously named `overlayfs` snapshotter path. If a file system access to the host machine is possible during a forensic live investigation, an analyst could retrieve or even place files from that mounted directory without having to connect to the executed container via the *Kubernetes* API using `kubectl` or other tools. This is especially useful in scenarios where the connection to a cluster is not functional anymore.

In conclusion for *containerd*, the files `meta.db` and `metadata.db` should be saved first in a forensic investigation. Using a parsing tool like *container-explorer*, they can be examined even at a later point in time. The `syslog` file and corresponding log-rotated versions should also be saved, as both *containerd* and *Kubernetes* store valuable information in it. If possible, the whole installation directory of *containerd* should be saved in general, as this would also allow to restore the snapshots which are mounted into

the containers and to examine the metadata store in detail. In a case where both databases are not available any longer or are corrupted, an analyst could still reconstruct container images from the files stored in the various sub-directories. Furthermore, a snapshot of the mounted file systems should be taken by using `findmnt` or a similar program, in order to not lose the information of mounted file systems. The *containerd*-related information found in the *ForensicArtifacts/artifacts* are in line with identified artifacts resulting from the analysis done in Chapter 5.

When it comes to *Kubernetes*, most of the information on the host is provided through the *kubelet*. The installation path of that resource is `/var/lib/kubelet` and contains configuration files, plugins, certificates and keys used for the PKI of *Kubernetes* and sub-directories that host *Pod*-specific information. The sub-directory `Pods/` contains further directories named after a *Pod*'s identifier. Each of those directories again follows the same structure and has the directories `containers/`, `plugins/` and `volumes`. Sub-directories of `containers/` are unfortunately not named after the identifier of that container like its done in the respective *containerd* directory. Instead, the name of the container is taken as specified in the *Pod*'s manifest file. When a deployment of a *Pod* is requested, the corresponding manifest file gets tracked in several locations. One of those is the `syslog` log file. The results in Section 5.4.3 show that both the deployment and the deletion of a *Pod* resource can be seen in that file, providing an analyst not only the specified options for the deployment but also timestamps of the operations. The particular container directory in `containers/` contains a file named with an identifier that is eight characters long. Searching for this name as done in Section 5.2.3 revealed that this file gets actually mounted into a container at path `/dev/termination-log` and will contain log information when a container gets terminated. The output, error and input information of containers are logged specifically at a path that follows the following naming: `/var/log/pods/<namespace>_<pod_name>_<pod_id>/<container_name>/<num>.log`. During the analysis of scenario 2, it was observed that only the output information of commands are logged, which have been specified directly in the *Pod*'s manifest file. Using `kubectl exec` to invoke a script did not produce output in the corresponding container's log file. This needs further testing in order to find out under which circumstances the output, error or input information gets redirected to the named file. The directory `/var/lib/kubelet/pods/pod_id/containers` furthermore contains a sub-directory which hosts the *storage volume* or storage objects which are mounted into the container. The type of the storage object can be identified by the naming of the sub-directory of that folder. In that, the folder name `volumes/kube`

`rnetes.io~projected/` identifies a projected volume while `volumes/kubernetes.io~configmap/` indicates a ConfigMap resource (Kubernetes, 2021q). This information is valuable to reconstruct what kind of storage objects were mounted into a *Pod*. Furthermore, it is important to clarify that *storage volume* is provisioned at *Pod* level but mounted into a running container. The fact that the storage location is independent from the container directories enables to make that storage persistent. All containers of a *Pod* may terminate, but the storage is still available if configured correspondingly. Cluster-related information of *Kubernetes* is stored in the *etcd Pod*. It is located on the master server at path `/var/lib/etcd/member/snap/db` by default but can be distributed across the cluster nodes when following a high availability scenario. The *etcd* consists of a key-value database which uses *bolt* as the underlying database component. The database was retrieved during the analysis of scenario 1. Using *etcd-dump-db* as parsing tool, the contents of that database were successfully investigated. It revealed the cluster's state at different points in time so that an analyst could reconstruct a cluster's layout using the information from that file alone. Correlating the retrieved information from the *etcd* database with the logging information from the `syslog` file could also help to trace back changes done within the cluster.

When in a hurry, an analyst should try to first save the *etcd* database and the `syslog` file, if possible. Those two files can be correlated to collect a detailed description of a cluster's state. If applicable, all directories containing *Kubernetes*-related artifacts should be saved recursively to enrich the information with those files.

The named *Kubernetes* artifacts have been documented following the YAML style definition of *ForensicArtifacts/artifacts*. The corresponding file containing the information can be found in Appendix A.1.7. The contribution has also been provided as a pull request to the corresponding upstream *Github* repository².

²see <https://github.com/ForensicArtifacts/artifacts/pull/444>

CONCLUSION AND FUTURE WORK

7.1. Future Work

This work has shown that using the approach of a differential analysis is also applicable to distributed virtual environments like a *Kubernetes* cluster. The developed *PowerShell* modules that enabled the automated analysis of multiple virtual machines could also be used for analysis independent from *Kubernetes* or even the used operating systems. To do so, the implementation needs to be tested on other environments analysing other applications and systems. A valuable enhancement could be to include a system's memory dump in the differential analysis, which would allow comparing running processes in between two states. Besides that, the network traffic of a node might also get included in the investigation. The file system mount points, for example, were retrieved by invoking `findmnt` on a node, saving the output on the analysis machine and normalizing it for further analysis. The same approach could be followed for analysing the network communication by invoking `tcpdump` on a node and capturing the outputs. Furthermore, the framework should be enhanced with the capability of automatically parsing files out of an exported image in order to compare them, too. One example for that could be to extract simple text files like the `syslog` file from the images of *Init* and *Action*, normalize them by only including the component and the actual message and then compare the files to each other in order to find characteristic log messages. Additionally, the amount of performed runs per action should be increased to make the results more robust. Within this work, each action has been performed around 20 times. Arguably, an action could be performed 1000 times and maybe an evidence that occurs 18 out of 20 times will only occur 800 out of 1000 times, which questions the overall robustness of such an evidence. The

developed framework will be made publicly available on *Github* so that the development of enhancements can be community-driven.

Furthermore, the cluster setup could be changed in order to analyze other container runtimes than *containerd* together with *Kubernetes*. Any other supported *CRI* implementation could be used instead (Kubernetes, 2021f). One question regarding such a scenario would be if and how the *Kubernetes* artifacts would change. It is also possible to exchange the container orchestration tool, so that, instead of *Kubernetes*, a framework like *OpenStack* gets installed and analyzed with the provided framework.

As this work focused highly on the actions that are related to *Kubernetes Pods*, further resources as described in Section 3.3.1 could get deployed in the cluster and analyzed following the same approach. This would be something that can be achieved by a community-driven effort, given that the tool set is built and working. Resulting artifacts might as well be contributed to the community as done in Chapter 6. Particularly, the *Kubernetes* API needs to be more integrated into the analysis process. The *Kubernetes* auditing feature, for example, provides an in-depth insight into the API communication in between the cluster components as well as traffic coming from or leaving the cluster (Kubernetes, 2021b). It should be evaluated whether those audit logs can also be integrated in the given analysis framework and how they differ from the `syslog`.

A more dedicated focus on productive infrastructures is also possible. One could set up a cluster that hosts a specific application with all its components in distributed namespaces and try to attack the application or *Kubernetes*. The attack could be seen as *Action* phase and thus the characteristic evidences would be related to a performed attack. The identified evidences or generic information could be used to harden the systems against such attacks, thus combining methods of offensive security testing and forensics.

All evidences identified within this work were found because full control over the nodes that hosted the *Kubernetes* cluster was given. This changes when a managed *Kubernetes* service like *AKS*, *EKS* or *GKE* is used. The main question would be if the identified artifacts would even be accessible, because in some offerings a node level access to the cluster is not planned at all. In such a case, it needs to be evaluated if and how the artifacts could be extracted from inside the *Kubernetes* cluster.

7.2. Conclusion

With the analytic approach used in this thesis, many forensic valuable artifacts were identified and described. Although the main objective was to identify artifacts related to *Kubernetes*, most of the evidences were resulting from and thus related to the used container runtime *containerd*. Nonetheless, the correlation between components in *containerd*, in *Kubernetes* and in between those two products has been worked out which provides a better understanding for analysts where to begin in a forensic investigation. It also helps an administrator to better understand what the structure of distributed container environments looks like and where the crucial information is stored. Especially the identified databases are worth being backed up regularly, either for the case of a corrupted infrastructure or to have them available for investigations. The same applies to the `syslog` file, which should be included in log-forwarding scenarios like Security information and event management systems, so that an alerting could be done for *containerd* or *Kubernetes* without actually having to rely on those components. Given the ephemeral nature of containers, it is arguably hard to decide against a complete redeployment of a cluster in case of a breach. But, as the analysis revealed and as stated in Chapter 6, an investigation for both *Kubernetes* and *containerd* can be performed without actually using the products. Instead of using `kubectl` in order to connect to a *Pod* and thus generating traffic in the *Kubernetes* API, an analyst could connect to the node hosting the *kubelet* directly via a remote shell and perform the analysis on file system level. This is a huge advantage for cases where an adversary scans the API communication to be aware of analysts, or, alternatively, in a case where using the API to interact with the *Kubernetes* cluster is not even possible anymore.

This said, a forensically sound investigation can be performed for both *containerd* and *Kubernetes* on file system level, both post-mortem and live.

BIBLIOGRAPHY

- Carrier, B. (2011). *The Sleuth Kit*. Version 6128426. The Sleuth Kit. URL: <https://github.com/sleuthkit/sleuthkit> (visited on 05/16/2021).
- Carrier, B. (2021). *Autopsy*. The Sleuth Kit. URL: <https://www.sleuthkit.org/autopsy/> (visited on 11/21/2021).
- Castle, G. and J. Metz (2014). *Digital Forensics Artifacts Repository*. Version 21ecc98. Forensic Artifacts. URL: <https://github.com/ForensicArtifacts/artifacts> (visited on 04/29/2021).
- CIS (2020). *CIS Benchmarks*. Center for Internet Security. URL: <https://www.cisecurity.org/cis-benchmarks/> (visited on 04/29/2021).
- CNCF (2014). *gRPC - An RPC library and framework*. Version 83868b6. Cloud Native Computing Foundation. URL: <https://github.com/grpc/grpc> (visited on 11/06/2021).
- CNCF (2021). *Building sustainable ecosystems for cloud native software*. Cloud Native Computing Foundation. URL: <https://www.cncf.io/> (visited on 04/29/2021).
- containerd (2015a). *An open and reliable container runtime*. Version d418660. Cloud Native Computing Foundation. URL: <https://github.com/containerd/containerd> (visited on 11/06/2021).
- containerd (2015b). *Architecture*. Version 3c5b0dc. Cloud Native Computing Foundation. URL: <https://github.com/containerd/containerd/blob/main/design/architecture.md> (visited on 11/06/2021).
- containerd (2015c). *Architecture of The CRI Plugin*. Version a05fa42. Cloud Native Computing Foundation. URL: <https://github.com/containerd/containerd/blob/main/docs/cri/architecture.md> (visited on 11/06/2021).

- containerd (2015d). *Container Lifecycle*. Version 4700968. Cloud Native Computing Foundation. URL: <https://github.com/containerd/containerd/blob/main/design/lifecycle.md> (visited on 11/06/2021).
- containerd (2015e). *containerd Plugins*. Version 6d3d34b. Cloud Native Computing Foundation. URL: <https://github.com/containerd/containerd/blob/e00f87f1dc6a22d3c34495dcd75769d80e56a761/docs/PLUGINS.md> (visited on 11/17/2021).
- containerd (2015f). *Data Flow*. Version a3cae91. Cloud Native Computing Foundation. URL: <https://github.com/containerd/containerd/blob/main/design/data-flow.md> (visited on 11/06/2021).
- containerd (2015g). *Getting started with containerd*. Version ebc5cf1. Cloud Native Computing Foundation. URL: <https://github.com/containerd/containerd/blob/main/docs/getting-started.md> (visited on 11/18/2021).
- Dewald, A. (2012). “Formalisierung digitaler Spuren und ihre Einbettung in die Forensische Informatik”. Erlangen, Universität Erlangen-Nürnberg, Diss., 2012. Erlangen: Universitätsbibliothek der Universität Erlangen-Nürnberg. URL: <http://www.opus.sub.uni-erlangen.de/opus/volltexte/2012/3943/>.
- Dewald, A. (2015). “Characteristic Evidence, Counter Evidence and Reconstruction Problems in Forensic Computing”. In: *2015 Ninth International Conference on IT Security Incident Management & IT Forensics*. IEEE, pp. 77–82. ISBN: 978-1-4799-9902-6. DOI: 10.1109/IMF.2015.15.
- Docker (2021a). *Accelerate how you build, share and run modern applications*. Docker. URL: <https://www.docker.com/> (visited on 04/29/2021).
- Docker (2021b). *busybox: Busybox base image*. Docker. URL: https://hub.docker.com/_/busybox (visited on 11/18/2021).
- Docker (2021c). *Docker Desktop WSL 2 backend*. Version c5f38d8. Docker. URL: <https://github.com/docker/docker.github.io/blob/master/desktop/windows/wsl.md> (visited on 11/09/2021).
- etcd (2013a). *API reference*. Version 138926b. Cloud Native Computing Foundation. URL: https://etcd.io/docs/v3.5/dev-guide/api_reference_v3/ (visited on 11/19/2021).
- etcd (2013b). *etcd: A distributed, reliable key-value store for the most critical data of a distributed system*. Cloud Native Computing Foundation. URL: <https://etcd.io/> (visited on 11/01/2021).

- etcd (2013c). *etcd: Distributed reliable key-value store for the most critical data of a distributed system*. Version 7e0248b. Cloud Native Computing Foundation. URL: <https://github.com/etcd-io/etcd> (visited on 11/18/2021).
- Garfinkel, S. (2012a). *DFXML*. Version 327a7f8. URL: <https://github.com/sim-song/dfxml> (visited on 04/29/2021).
- Garfinkel, S. (2012b). “Digital forensics XML and the DFXML toolset”. In: *Digital Investigation* 8.3-4, pp. 161–174. ISSN: 17422876. DOI: 10.1016/j.diin.2011.11.002.
- Garfinkel, S., A. J. Nelson, and J. Young (2012). “A general strategy for differential forensic analysis”. In: *Digital Investigation* 9, pp. 50–59. ISSN: 17422876. DOI: 10.1016/j.diin.2012.05.003.
- Garfinkel, S. L. (2009). “Automating Disk Forensic Processing with SleuthKit, XML and Python”. In: *2009 Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering*. IEEE, pp. 73–84. ISBN: 978-0-7695-3792-4. DOI: 10.1109/SADFE.2009.12.
- Google (2013). *GRR Rapid Response: Remote live forensics for incident response*. Version 6942fa3. Google. URL: <https://github.com/google/grr> (visited on 04/29/2021).
- Google (2014). *Timesketch: Collaborative forensic timeline analysis*. Version 0c7244c. Google. URL: <https://github.com/google/timesketch> (visited on 04/29/2021).
- Google (2015). *Turbinia: Automation and Scaling of Digital Forensics Tools*. Version 957c02e. Google. URL: <https://github.com/google/turbinia> (visited on 04/29/2021).
- Google (2020). *Cloud Forensics Utils: Python library to carry out DFIR analysis on the Cloud*. Version d15bc2a. Google. URL: <https://github.com/google/cloud-forensics-utils> (visited on 04/29/2021).
- Google (2021). *Container-Explorer*. Version 0482d1b. Google. URL: <https://github.com/google/container-explorer> (visited on 11/15/2021).
- Grunert, S. (2019). *Demystifying containers: part I: Kernel Space*. Cloud Native Computing Foundation. URL: <https://www.cncf.io/blog/2019/06/24/demystifying-containers-part-i-kernel-space/> (visited on 07/28/2021).
- HashiCorp (2010a). *HashiCorp Vagrant: Development Environments Made Easy*. HashiCorp. URL: <https://www.vagrantup.com/> (visited on 11/21/2021).
- HashiCorp (2010b). *Vagrant*. Version c0338b6. HashiCorp. URL: <https://github.com/hashicorp/vagrant> (visited on 11/21/2021).

- HashiCorp (2010c). *Vagrant and Windows Subsystem for Linux*. Version 2c3397c. HashiCorp. URL: <https://github.com/hashicorp/vagrant/blob/main/website/content/docs/other/wsl.mdx> (visited on 11/09/2021).
- Hasselbring, W. and G. Steinacker (2017). “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, pp. 243–246. ISBN: 978-1-5090-4793-2. DOI: 10.1109/ICSAW.2017.11.
- io, etcd (2017). *bbolt*. Version b18879e. Cloud Native Computing Foundation. URL: <https://github.com/etcd-io/bbolt> (visited on 11/14/2021).
- Kalber, S., A. Dewald, and F. C. Freiling (2013). “Forensic Application-Fingerprinting Based on File System Metadata”. In: *2013 Seventh International Conference on IT Security Incident Management and IT Forensics*. IEEE, pp. 98–112. ISBN: 978-1-4673-6307-5. DOI: 10.1109/IMF.2013.20.
- Kubernetes (2016). *Introducing Container Runtime Interface (CRI) in Kubernetes*. Cloud Native Computing Foundation. URL: <https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/> (visited on 11/06/2021).
- Kubernetes (2021a). *API OVERVIEW*. Version v1.19.0. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19> (visited on 11/18/2021).
- Kubernetes (2021b). *Auditing*. Version 108149fa2. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/tasks/debug-application-cluster/audit/> (visited on 05/11/2021).
- Kubernetes (2021c). *Authenticating*. Version f3921c902. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/reference/access-authn-authz/authentication/> (visited on 11/20/2021).
- Kubernetes (2021d). *Cluster Networking*. Version 33c363a37. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/cluster-administration/networking/> (visited on 11/02/2021).
- Kubernetes (2021e). *ConfigMaps*. Version 8c9c9c543. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/configuration/configmap/> (visited on 11/02/2021).
- Kubernetes (2021f). *Container runtimes*. Version a5b097977. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.19> (visited on 11/26/2021).

- Kubernetes (2021g). *Install Tools: kubectl*. Version e9703497a. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/tasks/tools/#kubectl> (visited on 11/21/2021).
- Kubernetes (2021h). *Kubernetes Components*. Version dc84f0cb9. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/overview/components/> (visited on 11/02/2021).
- Kubernetes (2021i). *Kubernetes: Production-Grade Container Orchestration*. Cloud Native Computing Foundation. URL: <https://kubernetes.io/> (visited on 04/29/2021).
- Kubernetes (2021j). *Logging Architecture*. Version f945335af. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/cluster-administration/logging/> (visited on 11/20/2021).
- Kubernetes (2021k). *Namespaces*. Version b3ff00304. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/> (visited on 11/18/2021).
- Kubernetes (2021l). *Persistent Volumes*. Version 9fe3e942f. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/storage/persistent-volumes/> (visited on 11/02/2021).
- Kubernetes (2021m). *Pods*. Version dc84f0cb9. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/workloads/pods/> (visited on 11/02/2021).
- Kubernetes (2021n). *ReplicaSet*. Version 5b373f5bb. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/> (visited on 11/02/2021).
- Kubernetes (2021o). *Secrets*. Version 97c35ce77. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/configuration/secret/> (visited on 11/02/2021).
- Kubernetes (2021p). *Service*. Version 7dd48726d. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/services-networking/service/> (visited on 11/02/2021).
- Kubernetes (2021q). *Volumes*. Version f945335af. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/storage/volumes/> (visited on 11/02/2021).
- Kubernetes (2021r). *What is Kubernetes?* Version eb90fddd. Cloud Native Computing Foundation. URL: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/> (visited on 11/02/2021).

- LeClaire, N. (2020). *Overcoming Infrastructure Obstacles When Deploying Production-Ready Kubernetes*. Dell Technologies and VMWare. URL: <https://www.delltechnologies.com/en-ca/collaterals/unauth/white-papers/products/converged-infrastructure/oreilly-paper-overcoming-infrastructure-obstacles-when-deploying-production-ready-kubernetes.pdf> (visited on 05/18/2021).
- library, docker (2015). *Docker Official Image packaging for Busybox*. Version 1b93708. Docker. URL: <https://github.com/docker-library/busybox> (visited on 11/06/2021).
- Linux Kernel Organization (2021a). *Overlay Filesystem*. URL: <https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html> (visited on 11/14/2021).
- Linux Kernel Organization (2021b). *The Linux Kernel Archives*. URL: <https://www.kernel.org/> (visited on 07/28/2021).
- Linux man-pages project (2019). *Linux Programmer's Manual: ipc_namespaces - overview of Linux IPC namespaces*. Version 5.13. URL: https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html (visited on 07/27/2021).
- Linux man-pages project (2020a). *Linux Programmer's Manual: cgroup_namespaces - overview of Linux cgroup namespaces*. Version 5.13. URL: https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html (visited on 09/28/2021).
- Linux man-pages project (2020b). *Linux Programmer's Manual: mq_overview - overview of POSIX message queues*. Version 5.13. URL: https://man7.org/linux/man-pages/man7/uts_namespaces.7.html (visited on 07/27/2021).
- Linux man-pages project (2020c). *Linux Programmer's Manual: sysvipc - System V inter-process communication mechanisms*. Version 5.12. URL: <https://man7.org/linux/man-pages/man7/sysvipc.7.html> (visited on 07/27/2021).
- Linux man-pages project (2021a). *Linux Programmer's Manual: capabilities - overview of Linux capabilities*. Version 5.13. URL: <https://man7.org/linux/man-pages/man7/capabilities.7.html> (visited on 09/28/2021).
- Linux man-pages project (2021b). *Linux Programmer's Manual: cgroups - Linux control groups*. Version 5.13. URL: <https://man7.org/linux/man-pages/man7/cgroups.7.html> (visited on 09/28/2021).

- Linux man-pages project (2021c). *Linux Programmer's Manual: chmod - change file mode bits*. URL: <https://man7.org/linux/man-pages/man1/chmod.1.html> (visited on 10/24/2021).
- Linux man-pages project (2021d). *Linux Programmer's Manual: clone, __clone2, clone3 - create a child process*. Version 5.12. URL: <https://man7.org/linux/man-pages/man2/clone.2.html> (visited on 07/28/2021).
- Linux man-pages project (2021e). *Linux Programmer's Manual: fork - create a child process*. Version 5.12. URL: <https://man7.org/linux/man-pages/man2/fork.2.html> (visited on 07/28/2021).
- Linux man-pages project (2021f). *Linux Programmer's Manual: mount_namespaces - overview of Linux mount namespaces*. Version 5.12. URL: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html (visited on 07/27/2021).
- Linux man-pages project (2021g). *Linux Programmer's Manual: namespaces - overview of Linux namespaces*. Version 5.12. URL: <https://man7.org/linux/man-pages/man7/namespaces.7.html> (visited on 07/27/2021).
- Linux man-pages project (2021h). *Linux Programmer's Manual: network_namespaces - overview of Linux network namespaces*. Version 5.13. URL: https://man7.org/linux/man-pages/man7/network_namespaces.7.html (visited on 07/27/2021).
- Linux man-pages project (2021i). *Linux Programmer's Manual: pid_namespaces - overview of Linux PID namespaces*. Version 5.12. URL: https://man7.org/linux/man-pages/man7/pid_namespaces.7.html (visited on 07/27/2021).
- Linux man-pages project (2021j). *Linux Programmer's Manual: proc - process information pseudo-filesystem*. Version 5.12. URL: <https://man7.org/linux/man-pages/man5/proc.5.html> (visited on 08/01/2021).
- Linux man-pages project (2021k). *Linux Programmer's Manual: setns - reassociate thread with a namespace*. Version 5.12. URL: <https://man7.org/linux/man-pages/man2/setns.2.html> (visited on 07/28/2021).
- Linux man-pages project (2021l). *Linux Programmer's Manual: systemd, init - systemd system and service manager*. Version 5.12. URL: <https://man7.org/linux/man-pages/man1/init.1.html> (visited on 08/01/2021).
- Linux man-pages project (2021m). *Linux Programmer's Manual: touch - change file timestamps*. URL: <https://man7.org/linux/man-pages/man1/touch.1.html> (visited on 10/24/2021).

- Linux man-pages project (2021n). *Linux Programmer's Manual: unshare - disassociate parts of the process execution context*. Version 5.12. URL: <https://man7.org/linux/man-pages/man1/unshare.1.html> (visited on 07/28/2021).
- Linux man-pages project (2021o). *Linux Programmer's Manual: user_namespaces - overview of Linux user namespaces*. Version 5.13. URL: https://man7.org/linux/man-pages/man7/user_namespaces.7.html (visited on 09/28/2021).
- Linux man-pages project (2021p). *Linux Programmer's Manual: uts_namespaces - overview of Linux UTS namespaces*. Version 5.12. URL: https://man7.org/linux/man-pages/man7/uts_namespaces.7.html (visited on 07/27/2021).
- Linux man-pages project (2021q). *Linux Programmer's Manual: veth - Virtual Ethernet Device*. Version 5.13. URL: <https://man7.org/linux/man-pages/man4/veth.4.html> (visited on 07/27/2021).
- log2timeline (2014). *Plaso: Super timeline all the things*. Version 509747a. log2timeline. URL: <https://github.com/log2timeline/plaso> (visited on 04/29/2021).
- log2timeline (2016). *DFTimewolf: A framework for orchestrating forensic collection, processing and data export*. Version 2bc7a7d. log2timeline. URL: <https://github.com/log2timeline/dftimewolf> (visited on 04/29/2021).
- OCI (2015a). *OCI Runtime Specification*. Version a3c33d6. Open Container Initiative. URL: <https://github.com/opencontainers/runtime-spec> (visited on 04/29/2021).
- OCI (2015b). *OCI Runtime Specification: Configuration*. Version c83b45e. Open Container Initiative. URL: <https://github.com/opencontainers/runtime-spec/blob/main/config.md> (visited on 11/19/2021).
- OCI (2015c). *runc: CLI tool for spawning and running containers according to the OCI specification*. Version ec9dc966. Open Container Initiative. URL: <https://github.com/opencontainers/runc> (visited on 11/06/2021).
- OCI (2016a). *Image Format Specification*. Version a25f547. Open Container Initiative. URL: <https://github.com/opencontainers/image-spec/blob/main/spec.md> (visited on 11/05/2021).
- OCI (2016b). *Image Layer Filesystem Changeset*. Version 1a29e86. Open Container Initiative. URL: <https://github.com/opencontainers/image-spec/blob/main/layer.md> (visited on 11/05/2021).
- OCI (2016c). *OCI Content Descriptors*. Version 2062c45. Open Container Initiative. URL: <https://github.com/opencontainers/image-spec/blob/main/descriptor.md> (visited on 11/05/2021).

- OCI (2016d). *OCI Image Configuration*. Version fc4df0a. Open Container Initiative. URL: <https://github.com/opencontainers/image-spec/blob/main/config.md> (visited on 11/05/2021).
- OCI (2016e). *OCI Image Format*. Version 9a7a987. Open Container Initiative. URL: <https://github.com/opencontainers/image-spec> (visited on 04/29/2021).
- OCI (2016f). *OCI Image Index Specification*. Version 170393e. Open Container Initiative. URL: <https://github.com/opencontainers/image-spec/blob/main/image-index.md> (visited on 11/05/2021).
- OCI (2016g). *OCI Image Manifest Specification*. Version 069d186. Open Container Initiative. URL: <https://github.com/opencontainers/image-spec/blob/main/manifest.md> (visited on 11/05/2021).
- OCI (2021). *Open Container Initiative*. Open Container Initiative. URL: <https://opencontainers.org/> (visited on 11/04/2021).
- Oracle (2021). *VirtualBox*. Oracle. URL: <https://www.virtualbox.org/> (visited on 04/29/2021).
- RedHat (2012). *Ansible*. Version d8aeffc. RedHat. URL: <https://github.com/ansible/ansible> (visited on 11/21/2021).
- RedLock (2018). *Lessons from the Cryptojacking Attack at Tesla*. RedLock. URL: <https://redlock.io/blog/cryptojacking-tesla> (visited on 05/19/2021).
- Riadi, I., R. Umar, and A. Sugandi (2020). “Web Forensic On Kubernetes Cluster Services Using Grr Rapid Response Framework”. In: *International Journal of Scientific & Technology Research* 09.01, pp. 3484–3488. ISSN: 2277-8616. URL: <http://www.ijstr.org/final-print/jan2020/Web-Forensic-On-Kubernetes-Cluster-Services-Using-Grr-Rapid-Response-Framework.pdf> (visited on 04/29/2021).

A

APPENDIX

A.1. Supplements

This section contains information about several supplementary files or directories stored on an optical data drive handed together with this thesis. All data paths are relative to the root directory of that drive. A list of checksums is located at the root directory.

A.1.1. PowerShell module: VBox4Pwsh

The directory `PowerShell\Modules\VBox4Pwsh` contains the sources that form the *PowerShell* module *VBox4Pwsh* which has been used throughout the analysis scenarios. It is also provided to the open source community and can be found in the following *Github* repository:

<https://github.com/pr3l14t0r/VBox4Pwsh>

The files that are shipped with this thesis do match the commit ID `4c6f609`.

A.1.2. PowerShell module: PowerForensicator

The directory `PowerShell\Modules\PowerForensicator` contains the sources that form the *PowerShell* module *PowerForensicator* which has been used throughout the analysis scenarios. It is also provided to the open source community and can be found in the following *Github* repository:

<https://github.com/pr3l14t0r/PowerForensicator>

The files that are shipped with this thesis do match the commit ID 2fcb851.

A.1.3. Cluster Setup

The directory `ClusterProvisioning` contains the configuration as code of the *Kubernetes* cluster used within this work. The *Vagrantfile* can be used to automatically setup the needed environment. It will also configure the cluster once the virtual machines have been successfully created. The tool *Ansible* is used for that specific purpose. To use it, an installation of *VirtualBox*, *Vagrant* and *Ansible* is required on the host system. If necessary, the configuration of CPU and RAM usage might be changed in the *Vagrantfile*. The *Ansible* roles are stored within the directory `ClusterProvisioning\roles`.

To setup the cluster, open a shell and navigate to the directory where the *Vagrantfile* is located and invoke *Vagrant* using the command: `vagrant up`. That will start the provisioning process. The file `ClusterProvisioning\README.md` contains further details for the setup.

The source code is tracked in a repository at *Github* within the branch `master_thesis`:

<https://github.com/pr3l14t0r/ansible-vbox-vagrant-kubernetes>.

Path	<code>ClusterProvisioning\Vagrantfile</code>
SHA1	<code>CE342C465F8601DB56A417C08C0F059541C358B3</code>

Table A.1.: File Information

A.1.4. Display-ContainerVolumes.ps1

This file contains a snippet where the JSON output of `kubectl` is taken as input and parsed in order to get information about mounted volumes of a container or multiple containers.

Path	<code>PowerShell\Scripts\Display-ContainerVolumes.ps1</code>
SHA1	<code>236B50F71ACD28EAF0CB769DD9FAE7789C0F4723</code>

Table A.2.: File Information

A.1.5. Dockerfile: *etcd-dump-db*

This file contains the needed instructions for *Docker* to build a container image. The image will contain the already built tool *etcd-dump-db* (etcd, 2013c) and can be used for creating containers. To build the image open a shell, navigate to the folder where the `Dockerfile` is located and invoke the command:

```
docker build -t etcd-dump-db .
```

The image will be built and stored locally.

Path	Dockerfiles\etcd-dump-db\Dockerfile
SHA1	00002DC827778288779CACC2C56DF94C669DEABB

Table A.3.: File Information

A.1.6. Dockerfile: *container-explorer*

This file contains the needed instructions for *Docker* to build a container image. The image will contain the already built tool *container-explorer* (Google, 2021) which is used in Section 5.5. To build the image open a shell, navigate to the folder where the `Dockerfile` is located and invoke the command:

```
docker build -t container-explorer .
```

The image will be built and stored locally. Further instructions as well as the documentation how the exported image has been mounted can be found in the `readme.md` file.

Path	Dockerfiles\container-explorer\Dockerfile
SHA1	FE7F67C416CED94B0415236D130753BB694256D3
Path	Dockerfiles\container-explorer\readme.md
SHA1	7E124C8E8C8025B449D25476CF7E481F3F12B674

Table A.4.: File Information

A.1.7. Kubernetes artifacts

This file contains the formalised artifacts following the style guide of the *Github* repository *ForensicArtifacts/artifacts* (Castle and Metz, 2014).

This file has also been contributed to that project. The corresponding pull request can be found at URL:

<https://github.com/ForensicArtifacts/artifacts/pull/444>

Path	Kubernetes_Artifacts\kubernetes.yaml
SHA1	995DAD10AA717BFCAE2683F42976C9BB724BE880

Table A.5.: File Information

A.2. OCI Image specification examples

A.2.1. Example image configuration

```
1 {
2   "created": "2015-10-31T22:22:56.015925234Z",
3   "author": "Alyssa P. Hacker <alyspdev@example.com>",
4   "architecture": "amd64",
5   "os": "linux",
6   "config": {
7     "User": "alice",
8     "ExposedPorts": {
9       "8080/tcp": {}
10    },
11    "Env": [
12      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/j
13      ↪ sbin:/bin",
14      "FOO=oci_is_a",
15      "BAR=well_written_spec"
16    ],
17    "Entrypoint": [
18      "/bin/my-app-binary"
19    ],
20    "Cmd": [
21      "--foreground",
22      "--config",
23      "/etc/my-app.d/default.cfg"
24    ],
25    "Volumes": {
26      "/var/job-result-data": {},

```

```

26     "/var/log/my-app-logs": {}
27 },
28     "WorkingDir": "/home/alice",
29     "Labels": {
30         "com.example.project.git.url":
31             ↪ "https://example.com/project.git",
32         "com.example.project.git.commit":
33             ↪ "45a939b2999782a3f005621a8d0f29aa387e1d6b"
34     }
35 },
36     "rootfs": {
37         "diff_ids": [
38             "sha256:c6f988f4874bb0add23a778f753c65efe992244e148a1d2ec2a8b",
39             ↪ "664fb66bbd1",
40             "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfa",
41             ↪ "f10ace3c6ef"
42         ],
43         "type": "layers"
44     },
45     "history": [
46         {
47             "created": "2015-10-31T22:22:54.690851953Z",
48             "created_by": "/bin/sh -c #(nop) ADD file:a3bc1e842b69636f9df",
49             ↪ "5256c49c5374fb4eef1e281fe3f282c65fb853ee171c5 in",
50             ↪ "/"
51         },
52         {
53             "created": "2015-10-31T22:22:55.613815829Z",
54             "created_by": "/bin/sh -c #(nop) CMD [\"sh\"]",
55             "empty_layer": true
56         },
57         {
58             "created": "2015-10-31T22:22:56.329850019Z",
59             "created_by": "/bin/sh -c apk add curl"
60         }
61     ]
62 }

```

Listing A.1.: Example image configuration, taken from (OCI, 2016d)

A.2.2. Example image index

```
1 {
2   "schemaVersion": 2,
3   "manifests": [
4     {
5       "mediaType": "application/vnd.oci.image.manifest.v1+json",
6       "size": 7143,
7       "digest": "sha256:e692418e4cbaf90ca69d05a66403747baa33ee0880665"
8       ↪ "0b51fab815ad7fc331f",
9       "platform": {
10        "architecture": "ppc64le",
11        "os": "linux"
12      }
13    },
14    {
15      "mediaType": "application/vnd.oci.image.manifest.v1+json",
16      "size": 7682,
17      "digest": "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa"
18      ↪ "1f392e94a4333501270",
19      "platform": {
20        "architecture": "amd64",
21        "os": "linux"
22      }
23    }
24  ],
25  "annotations": {
26    "com.example.key1": "value1",
27    "com.example.key2": "value2"
28  }
29 }
```

Listing A.2.: Example image index, taken from (OCI, 2016f)

A.3. Example analysis scenario

A.3.1. Characteristic evidences of master node

```
1 home/vagrant mc 20
2 home/vagrant/00_PreInit.sh d 20
3 home/vagrant/02_FileMods.sh cr 20
4 home/vagrant/Documents mc 20
5 home/vagrant/Documents/access.me ca 20
6 home/vagrant/Documents/append.me mc 20
7 home/vagrant/Documents/change.me mc 20
8 home/vagrant/Documents/chmod.me c 20
9 home/vagrant/Documents/create.me cr 20
10 home/vagrant/Documents/delete.me dmc 20
11 home/vagrant/Documents/IhaveBeen.renamed rc 20
12 home/vagrant/Documents/modify.me mc 20
13 home/vagrant/Documents/touch.me mca 20
14 tmp/systemd-private-blfb9a2a894642209bcccfl4abf7f2b-logrotate.servic_
   ↪ e-Mg2dUh d
   ↪ 20
15 var/lib/dpkg/updates/0071 mc 20
16 var/lib/systemd/timers/stamp-e2scrub_all.timer mca 20
17 var/lib/systemd/timers/stamp-ua-messaging.timer mca 20
18 var/lib/ubuntu-advantage/messages mc 20
19 var/lib/ubuntu-release-upgrader/modules.symbols.bin mca 20
20 var/log/ubuntu-advantage.log mca 20
```

Listing A.3.: Characteristic evidences of master node in the example scenario

A.3.2. Characteristic evidences of worker node

```
1 home/vagrant mc 20
2 home/vagrant/00_PreInit.sh d 20
3 home/vagrant/02_FileMods.sh cr 20
4 home/vagrant/Documents mc 20
5 home/vagrant/Documents/access.me ca 20
6 home/vagrant/Documents/append.me mc 20
```

```
7  home/vagrant/Documents/change.me mc 20
8  home/vagrant/Documents/chmod.me c 20
9  home/vagrant/Documents/create.me cr 20
10 home/vagrant/Documents/delete.me dmc 20
11 home/vagrant/Documents/IhaveBeen.renamed rc 20
12 home/vagrant/Documents/modify.me mc 20
13 home/vagrant/Documents/touch.me mca 20
14 var/lib/dpkg/updates/0071 mc 20
15 var/lib/systemd/timers/stamp-e2scrub_all.timer mca 20
16 var/lib/systemd/timers/stamp-ua-messaging.timer mca 20
17 var/lib/ubuntu-advantage/messages mc 20
18 var/lib/ubuntu-release-upgrader/modules.symbols.bin mca 20
19 var/log/ubuntu-advantage.log mca 20
```

Listing A.4.: Characteristic evidences of worker node in the example scenario

A.3.3. Selected fileobject of *idiff* file

```

216 <fileobject delta:deleted_file="1" delta:changed_file="1"
    ↳ delta:modified_file="1">
217     <parent_object>
218         <inode>1576240</inode>
219     </parent_object>
220     <filename>home/vagrant/Documents/delete.me</filename>
221     <partition>2</partition>
222     <id>384</id>
223     <name_type>r</name_type>
224     <filesize delta:changed_property="1">0</filesize>
225     <alloc delta:changed_property="1">0</alloc>
226     <used>1</used>
227     <inode>1576243</inode>
228     <meta_type>1</meta_type>
229     <mode>416</mode>
230     <nlink delta:changed_property="1">0</nlink>
231     <uid>1000</uid>
232     <gid>1000</gid>
233     <mtime delta:changed_property="1">2021-11-07T07:40:49Z</mtime>
234     <ctime delta:changed_property="1">2021-11-07T07:40:49Z</ctime>
235     <atime>2021-11-06T23:58:02Z</atime>
236     <crtime>2021-11-06T23:58:02Z</crtime>
237     <dtime delta:changed_property="1">2021-11-07T07:40:49Z</dtime>
238     <byte_runs facet="data" delta:changed_property="1" />
239     <hashdigest type="md5" delta:changed_property="1" />
240     <hashdigest type="sha1" delta:changed_property="1" />
241     <delta:original_fileobject>
242         <parent_object>
243             <inode>1576240</inode>
244         </parent_object>
245         <filename>home/vagrant/Documents/delete.me</filename>
246         <partition>2</partition>
247         <id>384</id>
248         <name_type>r</name_type>
249         <filesize>38</filesize>
250         <alloc>1</alloc>
251         <used>1</used>
252         <inode>1576243</inode>
253         <meta_type>1</meta_type>
254         <mode>416</mode>

```

```
255     <nlink>1</nlink>
256     <uid>1000</uid>
257     <gid>1000</gid>
258     <mtime>2021-11-06T23:58:02Z</mtime>
259     <ctime>2021-11-06T23:58:02Z</ctime>
260     <atime>2021-11-06T23:58:02Z</atime>
261     <crtime>2021-11-06T23:58:02Z</crtime>
262     <byte_runs>
263         <byte_run len="38" img_offset="28471496704"
                ↪ fs_offset="25911922688" file_offset="0" />
264     </byte_runs>
265     <hashdigest type="md5">a66c78ee4j
                ↪ 0569d14356a2395788656c4</hashdigest>
266     <hashdigest type="sha1">6d68ea0dbcfabee475545j
                ↪ 0acc488c67ab1c087c4</hashdigest>
267     </delta:original_fileobject>
268 </fileobject>
```

Listing A.5.: Content of selected `idiff` file showing the file object `delete.me`

A.3.4. Selected entry of fileobject

Entry of fileobject `IhaveBeen.renamed` in a selected `idiff` file

```
373 <fileobject delta:renamed_file="1" delta:changed_file="1">
374     <parent_object>
375         <inode>1576240</inode>
376     </parent_object>
377     <filename delta:changed_property="1">home/vagrant/Documentsj
                ↪ /IhaveBeen.renamed</filename>
378     <partition>2</partition>
379     <id>392</id>
380     <name_type>r</name_type>
```

```

381     <filesize>29</filesize>
382     <alloc>1</alloc>
383     <used>1</used>
384     <inode>1576250</inode>
385     <meta_type>1</meta_type>
386     <mode>416</mode>
387     <nlink>1</nlink>
388     <uid>1000</uid>
389     <gid>1000</gid>
390     <mtime>2021-11-06T23:58:02Z</mtime>
391     <ctime delta:changed_property="1">2021-11-07T07:40:49Z</ctime>
392     <atime>2021-11-06T23:58:02Z</atime>
393     <crttime>2021-11-06T23:58:02Z</crttime>
394     <byte_runs>
395         <byte_run len="29" img_offset="28471156736"
396             ↪ fs_offset="25911582720" file_offset="0" />
397     </byte_runs>
398     <hashdigest
399         ↪ type="md5">f13d1ecf6bf07fc9ca94543aa578c77e</hashdigest>
400     <hashdigest type="sha1">81ec5e7262048b49fc90fb82698af_j
401         ↪ 0923bab6a28</hashdigest>
402     <delta:original_fileobject>
403         <parent_object>
404             <inode>1576240</inode>
405         </parent_object>
406         <filename>home/vagrant/Documents/rename.me</filename>
407         <partition>2</partition>
408         <id>391</id>
409         <name_type>r</name_type>
410         <filesize>29</filesize>
411         <alloc>1</alloc>
412         <used>1</used>
413         <inode>1576250</inode>
414         <meta_type>1</meta_type>
415         <mode>416</mode>
416         <nlink>1</nlink>
417         <uid>1000</uid>
418         <gid>1000</gid>
419         <mtime>2021-11-06T23:58:02Z</mtime>
420         <ctime>2021-11-06T23:58:02Z</ctime>
421         <atime>2021-11-06T23:58:02Z</atime>
422         <crttime>2021-11-06T23:58:02Z</crttime>

```

```
420         <byte_runs>
421             <byte_run len="29" img_offset="28471156736"
               ↳ fs_offset="25911582720" file_offset="0" />
422         </byte_runs>
423         <hashdigest type="md5">f13d1ecf6bf_
               ↳ 07fc9ca94543aa578c77e</hashdigest>
424         <hashdigest type="sha1">81ec5e7262048b49fc90fb82698af_
               ↳ 0923bab6a28</hashdigest>
425     </delta:original_fileobject>
426 </fileobject>
```

Listing A.6.: Content of selected `idiff` file showing the file object `IhaveBeen.renamed`

A.4. Analysis environment

Component	Windows	Ubuntu
Operating System	Microsoft Windows 10 Pro 10.0.19043	Ubuntu 20.04.3 LTS
<i>VirtualBox</i>	6.1.22r144080	6.1.28r147628
<i>Vagrant</i>	Vagrant 2.2.18 (in WSL2)	Vagrant 2.2.18
<i>Ansible</i>	ansible 2.10.12 (in WSL2)	ansible 2.9.6
<i>PowerShell</i>	PowerShell 7.1.3	PowerShell 7.2.0
<i>Docker</i>	Docker version 20.10.8, build 3967b7d	Docker version 20.10.10, build b485636
<i>kubectrl</i>	Client Version: version.Info{Major:"1", Minor:"21", GitVersion:"v1.21.5", GitCommit:"aea7bbadd2\~fc0cd689de94a54e\~b7b758869d691", GitTreeState:"clean", BuildDate:"2021\~09\~15T21:10:45Z", GoVersion:"go1.16.8", Compiler:"gc", Platform:"windows /amd64"}	Client Version: version.Info{Major:"1", Minor:"22", GitVersion:"v1.22.3", GitCommit:"c920368204\~99fedefec0f847e2\~054d824aea6cd1", GitTreeState:"clean", BuildDate:"2021\~10\~27T18:41:28Z", GoVersion:"go1.16.9", Compiler:"gc", Platform:"linux /amd64"}

Table A.6.: Overview of software components used for the analysis

A.5. Scenario 01: Deployment of a Pod

This section contains information about the supplementary files included on an optical data drive handed together with this thesis. All data paths are relative to the root directory of that drive.

A.5.1. Analysis.ps1

This file contains the *PowerShell* script which automates the analysis performed in Section 5.2.

Path	Scenario1-DeployPod\Analysis_Files\Analysis.ps1
SHA1	22387845E6CA5EED1D098587BE5EE5AFA6AC185D

Table A.7.: File Information

A.5.2. simplePod.yaml

This file contains the description of a *Pod* resource which will be applied to the cluster within the *Action*-phase of Section 5.2.2.

Path	Scenario1-DeployPod\Analysis_Files\simplePod.yaml
SHA1	E407C86BADFE6C8CBDE00CBAA28CA27AFDB55EF7

Table A.8.: File Information

A.5.3. Result files - Ubuntu

The following table shows an overview of the files that resulted from the analysis taken in Section 5.2.2 on the *Ubuntu* machine. Their root location is Scenario1-DeployPod\Results\Kubernetes_DeployPod_Ubuntu.

Description	Characteristic evidences of file system analysis from master node.
Path	kubernetes-master-1-DeployPod-CharacteristicEvidence.txt
SHA1	2DA33179F280B21ECC64F3C05E71B5719FF6C6AD
Description	Characteristic evidences of file system analysis from worker node.
Path	kubernetes-worker-1-DeployPod-CharacteristicEvidence.txt
SHA1	F8105AF3215ED87132CB7AA9BABAF2E85900C7FA
Description	Characteristic evidences of mount analysis from worker node.
Path	kubernetes-worker-1-DeployPod-FSMounts-CharacteristicEvidence.txt
SHA1	DB2131F04B236F0EA0902809DC714967D2F6348F
Description	Log file of the analysis with verbose information.
Path	AnalysisLog.txt
SHA1	29C93EC38EC7FDC4FC6837232AEE5FA563C5EA21

Table A.9.: File Information

A.5.4. Result files - Windows

The following table shows an overview of the files that resulted from the analysis taken in Section 5.2.2 on the *Windows* machine. Their root location is Scenario1-DeployPod\Results\Kubernetes_DeployPod_Windows.

Description	Characteristic evidences of file system analysis from master node.
Path	kubernetes-master-1-DeployPod-CharacteristicEvidence.txt
SHA1	A5EA9B47A8AE2CDDDB8985C3E1D20245B941C00EC
Description	Characteristic evidences of file system analysis from worker node.
Path	kubernetes-worker-1-DeployPod-CharacteristicEvidence.txt
SHA1	BB13FC7F703B7D9BD0CE7A8F01B93FE818B1C0F5
Description	Characteristic evidences of mount analysis from worker node.
Path	kubernetes-worker-1-DeployPod-FSMounts-CharacteristicEvidence.txt
SHA1	34B57E9A94ED47F7DABF3CBFDA284D563911F7B8
Description	Log file of the analysis with verbose information.
Path	AnalysisLog.txt
SHA1	7C48F55D7A420B578F1FEE25590BF88B2FBE7658

Table A.10.: File Information

A.5.5. Exported files

The following directory contains files that have been exported from the raw image during the analysis of the results in Section 5.2.3. All exported files can be found in the directory:

Scenario1-DeployPod\Results\Exports

A.6. Scenario 02: Deploy crafted Container

This section contains information about the supplementary files included on an optical data drive handed together with this thesis. All data paths are relative to the root directory of that drive.

A.6.1. Analysis.ps1

This file contains the *PowerShell* script which automates the analysis performed in Section 5.3.

Path	Scenario2-DeployCraftedContainer\Analysis_Files\Analysis.ps1
SHA1	3FB8467C823917CDAE10851433057C0503EFD8B0

Table A.11.: File Information

A.6.2. Dockerfile

This file contains the instructions that *Docker* will parse to build an image. The image has already been uploaded to a public registry, therefore there is no need to build it locally in order to perform the investigation.

Path	Scenario2-DeployCraftedContainer\Analysis_Files\Dockerfile
SHA1	6E8A253FD75A0727ADE54CA93D67D400AC90614E

Table A.12.: File Information

A.6.3. craftedPod.yaml

This file contains the description of a *Pod* resource which will be applied to the cluster within the *Init*-phase of Section 5.3.2.

Path	Scenario2-DeployCraftedContainer\Analysis_Files\craftedPod.yaml
SHA1	0089347E8D546961B7A94E71D86A9FB757C52248

Table A.13.: File Information

A.6.4. DownloadFiles.sh

This file contains the description of a *Pod* resource which will be applied to the cluster within the *Init*-phase of Section 5.3.2.

Path	Scenario2-DeployCraftedContainer\Analysis_Files\DownloadFiles.sh
SHA1	705808C3EB82DD3EF060F6C07995536078C14A10

Table A.14.: File Information

A.6.5. Result files - Ubuntu

The following table shows an overview of the files that resulted from the analysis taken in Section 5.3.2 on the *Ubuntu* machine. Their root location is Scenario2-DeployCraftedContainer\Results\Kubernetes_DeployCraftedContainer_Ubuntu.

Description	Characteristic evidences of file system analysis from master node.
Path	kubernetes-master-1-StartDownload-CharacteristicEvidence.txt
SHA1	438C0A61ADDB439EE7EA0CB9BF47280E30B86206
Description	Characteristic evidences of file system analysis from worker node.
Path	kubernetes-worker-1-StartDownload-CharacteristicEvidence.txt
SHA1	80A4D123585FB69998236C8FE262D63E952A4820
Description	Characteristic evidences of mount analysis from worker node.
Path	kubernetes-worker-1-StartDownload-FSMounts-CharacteristicEvidence.txt
SHA1	6AE406162AE8B8A56A65AFC7D951D5C1CB17EE7C
Description	Log file of the analysis with verbose information.
Path	AnalysisLog.txt
SHA1	35448AB3A8CBAAC7F116C3F0F280972F50DAAE10

Table A.15.: File Information

A.6.6. Result files - Windows

The following table shows an overview of the files that resulted from the analysis taken in Section 5.3.2 on the *Windows* machine. Their root location is Scenario2-DeployCraftedContainer\Results\Kubernetes_DeployCraftedContainer_Windows.

Description	Characteristic evidences of file system analysis from worker node.
Path	kubernetes-worker-1-StartDownload-CharacteristicEvidence.txt
SHA1	F622663122860056E7634F3C8E02E183857CFFA1
Description	Characteristic evidences of mount analysis from worker node.
Path	kubernetes-worker-1-StartDownload-FSMounts-CharacteristicEvidence.txt
SHA1	62CC6F0E129DDAFFDB5CF226162990CC3390815E
Description	Log file of the analysis with verbose information.
Path	AnalysisLog.txt
SHA1	DDC4CE5A6A6FAE01F00DFF2C7EAE5565044C3B62
Description	This file contains the output of <code>findmnt</code> invoked within the <i>Action</i> -phase of round 20 of the analysis on the worker node.
Path	StartDownload\kubernetes-worker-1.StartDownload.20.FileSystemMounts.txt
SHA1	DDC4CE5A6A6FAE01F00DFF2C7EAE5565044C3B62

Table A.16.: File Information

A.6.7. Exported files

The following directory contains files that have been exported from the raw image during the analysis of the results in Section 5.3.3. All exported files can be found in the directory:

Scenario2-DeployCraftedContainer\Results\Exports

Description	This file contains the <i>manifest</i> of the used container image.
Path	e4624edaa6d5bcf871e35ab15ae231ecaa925d67ecdd93bc555ca6c5f7096f9a
SHA1	5A47786C198D8FD863839967CB52E5A60EAEA35C
Description	This file contains the <i>configuration</i> of the used container image.
Path	WorkerNode\84e90d9540723be712a71a8ed21f082847b89f444d9c9c8a9d2696682d1f9848
SHA1	WorkerNode\629FBD0CAE350D78B45CDF715F093DE20C62ACD4

Table A.17.: File Information

A.7. Scenario 3: Deletion of a *Pod*

This section contains information about the supplementary files included on an optical data drive handed together with this thesis. All data paths are relative to the root directory of that drive.

A.7.1. Analysis.ps1

This file contains the *PowerShell* script which automates the analysis performed in Section 5.4.

Path	Scenario3-DeletePod\Analysis_Files\Analysis.ps1
SHA1	AFDDBAD492EEE0B3AFC3C50899DC66DC99F8E577

Table A.18.: File Information

A.7.2. CraftedPod.yaml

This file contains the description of a *Pod* resource which will be applied to the cluster within the *Init*-phase of Section 5.4.2.

Path	Scenario3-DeletePod\craftedPod.yaml
SHA1	0089347E8D546961B7A94E71D86A9FB757C52248

Table A.19.: File Information

A.7.3. Result files - Ubuntu

The following table shows an overview of the files that resulted from the analysis taken in Section 5.4.2 on the *Ubuntu* machine. Their root location is Scenario3-DeletePod\Results\Kubernetes_DeletePod_UBUNTU.

Description	Characteristic evidences of file system analysis from master node.
Path	kubernetes-master-1-DeletePod-CharacteristicEvidence.txt
SHA1	1F365869C712A6C8DE055373D3262804D3727630
Description	Characteristic evidences of file system analysis from worker node.
Path	kubernetes-worker-1-DeletePod-CharacteristicEvidence.txt
SHA1	C8DDD595DF5B21E6E714961285146BEA3DCC480B
Description	Characteristic evidences of mount analysis from worker node.
Path	kubernetes-worker-1-DeletePod-FSMounts-CharacteristicEvidence.txt
SHA1	B06E496924DEA53EF209CB18D4B60046D49E0811
Description	Log file of the analysis with verbose information.
Path	AnalysisLog.txt
SHA1	5CB15BE582072D140A8577EF5FE62BBE2B60599C

Table A.20.: File Information

A.7.4. Result files - Windows

The following table shows an overview of the files that resulted from the analysis taken in Section 5.4.2 on the *Windows* machine. Their root location is Scenario3-DeletePod\Results\Kubernetes_DeletePod_WINDOWS.

Description	Characteristic evidences of file system analysis from master node.
Path	kubernetes-master-1-DeletePod-CharacteristicEvidence.txt
SHA1	55E832F97DECC44BD3407AFDE3C2638864827C38
Description	Characteristic evidences of file system analysis from worker node.
Path	kubernetes-worker-1-DeletePod-CharacteristicEvidence.txt
SHA1	335DC6DE98EC831AFE19F391ACE49039DE6D5ECF
Description	Characteristic evidences of mount analysis from worker node.
Path	kubernetes-worker-1-DeletePod-FSMounts-CharacteristicEvidence.txt
SHA1	88F0FE6030B27983E6C21409AFBFB399F48181CE
Description	Log file of the analysis with verbose information.
Path	AnalysisLog.txt
SHA1	6E45F07E9AB1B52F71C734F17DD8D5F9CEC03433

Table A.21.: File Information

A.7.5. Exported files

The following directory contains files that have been exported from the raw image during the analysis of the results in Section 5.4.3. All exported files can be found in the directory:

Scenario3-DeletePod\Results\Exports


Statutory Declaration

Statutory Declaration

I hereby declare that I have composed the present thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published.

Ehrenwörtliche Erklärung

Hiermit versichere ich ehrenwörtlich, dass ich die vorliegende Studienarbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.



Christoph Lünswilken

Hannover, 27.11.2021

Place and Date